目錄

- Introduction 1.1
- 简介 1.2
- 智能合约介绍 1.3
- 安装Solidity 1.4
- Solidity 编程实例 1.5
- 深入理解 Solidity 1.6
  - 源文件的布局 1.6.1
  - 合约的结构 1.6.2
  - 类型 1.6.3
  - 单位和全局可用变量 1.6.4
  - 表达式和控制结构 1.6.5
  - 合约 1.6.6
  - 杂项 1.6.7
- Security Considerations 1.7
- 编程规范 1.8
- 通用模式 1.9
- 常见问题 1.10

Solidity 官方文档中文版

欢迎各路区块链及以太坊领域的专家和爱好者参与这一翻译项目，我们会为每位翻译和校对人员署名。

特别介绍

Solidity 官方文档前一版已经翻译完成90%,内容需要更新。

前一版贡献者名单：

- 少平
- abao
- 志伟
- jinfengbc
- 大青蛙
- zhangyaning

一起来参与

如果想做出贡献(翻译或者校对)的话，请加QQ群：195819423，谢谢！

PS: 想探讨Solidity技术的可以加"Solidity技术交流群"：549293015

内容来源

英文官方网站：
https://solidity.readthedocs.io/

官方GitHub仓库：
https://github.com/ethereum/solidity

中文版 GitHub 仓库：
https://github.com/twq0076262/solidity-zh

参与步骤

- fork主仓库（https://github.com/twq0076262/solidity-zh）
- 按照章节认领翻译(每次申请一个章节)或者校对(可申请多个章节)在下面这

个 `README.md` 里找还没有被人申请的章节，写上（@你的github号），给主仓库的 `master` 分支提pull request；
- 提的 pull request 被确认，合并到主仓库后，代表你申请的章节认领完成，开始翻译或校对；
- 翻译或校对的文件为README.md或者TOC.md中对应的md文件，请不要翻译/校对单独文件夹中的index.md
- 翻译过程请参照 翻译协作规范 (见下一节)，完成翻译后提交 pull request 给主仓库的 `master` 分支；
- 完成校对后同样提交 pull request 给主仓库的 `master` 分支；
- 全部翻译/校对完成后，我们会生成PDF/ePub文档，放在极客学院Wiki平台发布，并为所有参与者署名！

翻译协作规范

为了让大家协作顺畅，需要每一个人遵循如下协作规范~

- 如果对Markdown和GitHub不了解，请先阅读如何使用Markdown以及如何使用GitHub
- 使用Markdown进行翻译，文件名必须使用英文
- 翻译后的文档请放到SOURCE文件夹下的对应章节中，然后pull request即可
- 如遇到文中的图片，请统一放在SOURCE/images目录下
- 原文中的HTML标签及代码请不要修改、翻译
- 有其他任何问题都欢迎发issue，我们看到了会尽快回复
- 翻译人员需将对应的原文地址和翻译人姓名添加到译文末尾，审校人员需要将自己的名字添加到译文末尾，具体格式请参见样例：

> 原文：Color Palettes 翻译：iceskysl 校对：PoppinLp

校对规范

- 认领校对时请提供相关方面的专业背景说明，保证校对质量
- 文章通顺，适合人类阅读与理解，别让人以为是机翻
- 确保图片都能正常显示，且其相对地址都是SOURCE/images，图片存放正确
- 专有名词符合术语表中的翻译要求，拿不准的新词汇可以使用中文翻译后加括号内英文的形式
- 校对人员需有一定专业背景，保证译文无专业知识方面错误
- 请查看翻译人员是否将原文链接和翻译链接放到译文最下方，审校人员需要将自己的名字添加到译文末尾，具体格式请参见样例：

> 原文：Color Palettes 翻译：iceskysl 校对：PoppinLp

- 有任何问题请提Issues或到协同翻译群讨论，校对后提PR等待Merge，管理员通过后会

> 在目录后面打√标识完成

参与者（按认领章节排序）

翻译 & 校对

- 智能合约介绍 翻译：（待认领）校对：(待认领)
- 安装Solidity 翻译：（待认领）校对：(待认领)
- Solidity 编程实例 翻译：（待认领）校对：(待认领)
- 深入理解 Solidity 翻译：（待认领）校对：(待认领)
    - 源文件的布局 翻译：（待认领）校对：(待认领)
    - 合约的结构 翻译：（待认领）校对：(待认领)
    - 类型 翻译：（待认领）校对：(待认领)
    - 单位和全局可用变量 翻译：（待认领）校对：(待认领)
    - 表达式和控制结构 翻译：（待认领）校对：(待认领)
    - 合约 翻译：（待认领）校对：(待认领)
    - 杂项 翻译：（待认领）校对：(待认领)
- Security Considerations翻译：（待认领）校对：(待认领)
- 编程规范 翻译：（待认领）校对：(待认领)
- 通用模式 翻译：（待认领）校对：(待认领)
- 常见问题 翻译：（待认领）校对：(待认领)

进度记录

- 2016-8-26, 极客学院Wiki启动协同翻译，创建 GitHub 仓库，制定协同规范

感谢支持

离线版本

简介

Solidity是一种语法类似JavaScript的高级语言。它被设计成以编译的方式生成以太坊虚拟机代码。在后续内容中你将会发现，使用它很容易创建用于投票、众筹、封闭拍卖、多重签名钱包等等的合约。

注意

目前尝试Solidity的最好方式是使用基于浏览器的编译器（需要一点时间加载，请耐心等待）。

有用链接

- Ethereum
- Browser-Based CompilerChangelog
- Story Backlog
- Source Code
- Gitter Chat

Solidity 文档

在下一章中，我们先看一个用Solidity写的简单的智能合约，然后介绍一下区块链和以太坊虚拟机的基础知识。

后续章节会通过一些实用的合约例子，来探索Solidity的一系列特性。记住，你可以在浏览器中尝试这些合约。

最后以及更多扩展章节的内容，会深入到Solidity 的各个方面。

如有任何关于Solidiy，或者本文档的问题及改进建议，请在gitter频道提出来。

智能合约介绍

一个简单的智能合约

先从一个非常基础的例子开始，不用担心你现在还一点都不了解，我们将逐步了解到更多的细节。

Storage

```
contract SimpleStorage {
    uint storedData;

    function set(uint x) {
        storedData = x;
    }

    function get() constant returns (uint retVal) {
        return storedData;
    }
}
```

在Solidity中，一个合约由一组代码（合约的函数）和数据（合约的状态）组成。合约位于以太坊区块链上的一个特殊地址。*uint storedData**;* 这行代码声明了一个状态变量，变量名为*storedData*，类型为 *uint* （256bits无符号整数）。你可以认为它就像数据库里面的一个存储单元，跟管理数据库一样，可以通过调用函数查询和修改它。在以太坊中，通常只有合约的拥有者才能这样做。在这个例子中，函数 *set* 和 *get* 分别用于修改和查询变量的值。

跟很多其他语言一样，访问状态变量时，不需要在前面增加 this. 这样的前缀。

这个合约还无法做很多事情（受限于以太坊的基础设施），仅仅是允许任何人储存一个数字。而且世界上任何一个人都可以来存取这个数字，缺少一个（可靠的）方式来保护你发布的数字。任何人都可以调用set方法设置一个不同的数字覆盖你发布的数字。但是你的数字将会留存在区块链的历史上。稍后我们会学习如何增加一个存取限制，使得只有你才能修改这个数字。

代币的例子

接下来的合约将实现一个形式最简单的加密货币。空中取币不再是一个魔术，当然只有创建合约的人才能做这件事情（想用其他货币发行模式也很简单，只是实现细节上的差异）。而且任何人都可以发送货币给其他人，不需要注册用户名和密码，只要有一对以太坊的公私钥即可。

Note

对于在线solidity环境来说，这不是一个好的例子。如果你使用在线solidity环境 来尝试这个例子。调用函数时，将无法改变from的地址。所以你只能扮演铸币者的角色，可以铸造货币并发送给其他人，而无法扮演其他人的角色。这点在线solidity环境将来会做改进。

```
contract Coin {
//关键字“public”使变量能从合约外部访问。
    address public minter;
    mapping (address => uint) public balances;

//事件让轻客户端能高效的对变化做出反应。
    event Sent(address from, address to, uint amount);

//这个构造函数的代码仅仅只在合约创建的时候被运行。
    function Coin() {
        minter = msg.sender;
    }
    function mint(address receiver, uint amount) {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }
    function send(address receiver, uint amount) {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        Sent(msg.sender, receiver, amount);
    }
}
```

这个合约引入了一些新的概念，让我们一个一个来看一下。

`address public minter;` 这行代码声明了一个可公开访问的状态变量，类型为address。address类型的值大小为160 bits，不支持任何算术操作。适用于存储合约的地址或其他人的公私钥。public关键字会自动为其修饰的状态变量生成访问函数。没有public关键字的变量将无法被其他合约访问。另外只有本合约内的代码才能写入。自动生成的函数如下：

```
function minter() returns (address) { return minter; }
```

当然我们自己增加一个这样的访问函数是行不通的。编译器会报错，指出这个函数与一个状态变量重名。

下一行代码 `mapping (address => uint) public balances;` 创建了一个public的状态变量，但是其类型更加的复杂。该类型将一些address映射到无符号整数。mapping可以被认为是一个哈希表，每一个可能的key对应的value被虚拟的初始化为全0.这个类比不是很严谨，对于一个mapping，无法获取一个包含其所有key或者value的链表。所以我们得自己记着添加了哪些东西到mapping中。更好的方式是维护一个这样的链表，或者使用其他更高级的数据类型。或者只在不受这个缺陷影响的场景中使用mapping，就像这个例子。在这个例子中由public关键字生成的访问函数将会更加复杂，其代码大致如下：

```
function balances(address _account) returns (uint balance) {
    return balances[_account];
}
```

我们可以很方便的通过这个函数查询某个特定账号的余额。

`event Sent(address from, address to, uint value);` 这行代码声明了一个“事件”。由send函数的最后一行代码触发。客户端（服务端应用也适用）可以以很低的开销来监听这些由区块链触发的事件。事件触发时，监听者会同时接收到from，to，value这些参数值，可以方便的用于跟踪交易。为了监听这个事件，你可以使用如下代码：

```
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
}
```

注意在客户端中是如何调用自动生成的 balances 函数的。

这里有个比较特殊的函数 Coin。它是一个构造函数，会在合约创建的时候运行，之后就无法被调用。它会永久得存储合约创建者的地址。msg（以及tx和block）是一个神奇的全局变量，它包含了一些可以被合约代码访问的属于区块链的属性。msg.sender 总是存放着当前函数的外部调用者的地址。

最后，真正被用户或者其他合约调用，用来完成本合约功能的函数是mint和send。如果合约创建者之外的其他人调用mint，什么都不会发生。而send可以被任何人（拥有一定数量的代币）调用，发送一些币给其他人。注意，当你通过该合约发送一些代币到某个地址，在区块链浏览器中查询该地址将什么也看不到。因为发送代币导致的余额变化只存储在该代币合约的数据存储中。通过事件我们可以很容易创建一个可以追踪你的新币交易和余额的“区块链浏览器”。

区块链基础

对于程序员来说，区块链这个概念其实不难理解。因为最难懂的一些东西（挖矿，哈希，椭圆曲线加密，点对点网络等等）只是为了提供一系列的特性和保障。你只需要接受这些既有的特性，不需要关心其底层的技术。就像你如果仅仅是为了使用亚马逊的AWS，并不需要了解其内部工作原理。

交易/事务

区块链是一个全局共享的，事务性的数据库。这意味着参与这个网络的每一个人都可以读取其中的记录。如果你想修改这个数据库中的东西，就必须创建一个事务，并得到其他所有人的确认。事务这个词意味着你要做的修改（假如你想同时修改两个值）只能被完完全全的实施或者一点都没有进行。

此外，当你的事务被应用到这个数据库的时候，其他事务不能修改该数据库。

举个例子，想象一张表，里面列出了某个电子货币所有账号的余额。当从一个账户到另外一个账户的转账请求发生时，这个数据库的事务特性确保从一个账户中减掉的金额会被加到另一个账户上。如果因为某种原因，往目标账户上增加金额无法进行，那么源账户的金额也不会发生任何变化。

此外，一个事务会被发送者（创建者）进行密码学签名。这项措施非常直观的为数据库的特定修改增加了访问保护。在电子货币的例子中，一个简单的检查就可以确保只有持有账户密钥的人，才能从该账户向外转账。

区块

区块链要解决的一个主要难题，在比特币中被称为“双花攻击”。当网络上出现了两笔交易，都要花光一个账户中的钱时，会发生什么？一个冲突？

简单的回答是你不需要关心这个问题。这些交易会被排序并打包成“区块”，然后被所有参与的节点执行和分发。如果两笔交易相互冲突，排序靠后的交易会被拒绝并剔除出区块。

这些区块按时间排成一个线性序列。这也正是“区块链”这个词的由来。区块以一个相当规律的时间间隔加入到链上。对于以太坊，这个间隔大致是17秒。

作为“顺序选择机制”（通常称为“挖矿”）的一部分，一段区块链可能会时不时被回滚。但这种情况只会发生在整条链的末端。回滚涉及的区块越多，其发生的概率越小。所以你的交易可能会被回滚，甚至会被从区块链中删除。但是你等待的越久，这种情况发生的概率就越小。

以太坊虚拟机

总览

以太坊虚拟机（EVM）是以太坊中智能合约的运行环境。它不仅被沙箱封装起来，事实上它被完全隔离，也就是说运行在EVM内部的代码不能接触到网络、文件系统或者其它进程。甚至智能合约与其它智能合约只有有限的接触。

账户

以太坊中有两类账户，它们共用同一个地址空间。外部账户，该类账户被公钥-私钥对控制（人类）。合约账户，该类账户被存储在账户中的代码控制。

外部账户的地址是由公钥决定的，合约账户的地址是在创建该合约时确定的（这个地址由合约创建者的地址和该地址发出过的交易数量计算得到，地址发出过的交易数量也被称作"nonce"）

合约账户存储了代码，外部账户则没有，除了这点以外，这两类账户对于EVM来说是一样的。

每个账户有一个key-value形式的持久化存储。其中key和value的长度都是256比特，名字叫做storage.

另外，每个账户都有一个以太币余额（单位是“Wei"），该账户余额可以通过向它发送带有以太币的交易来改变。

交易

一笔交易是一条消息，从一个账户发送到另一个账户（可能是相同的账户或者零账户，见下文）。交易可以包含二进制数据（payload）和以太币。

如果目标账户包含代码，该代码会执行，payload就是输入数据。

如果目标账户是零账户（账户地址是0），交易将创建一个新合约。正如上文所讲，这个合约地址不是零地址，而是由合约创建者的地址和该地址发出过的交易数量（被称为nonce）计算得到。创建合约交易的payload被当作EVM字节码执行。执行的输出做为合约代码被永久存储。这意味着，为了创建一个合约，你不需要向合约发送真正的合约代码，而是发送能够返回真正代码的代码。

Gas

以太坊上的每笔交易都会被收取一定数量的gas，gas的目的是限制执行交易所需的工作量，同时为执行支付费用。当EVM执行交易时，gas将按照特定规则被逐渐消耗。

gas price（以太币计）是由交易创建者设置的，发送账户需要预付的交易费用 = gas price * gas amount。 如果执行结束还有gas剩余，这些gas将被返还给发送账户。

无论执行到什么位置，一旦gas被耗尽（比如降为负值），将会触发一个out-of-gas异常。当前调用帧所做的所有状态修改都将被回滚。

存储，主存和栈

每个账户有一块持久化内存区域被称为存储。其形式为key-value，key和value的长度均为256比特。在合约里，不能遍历账户的存储。相对于另外两种，存储的读操作相对来说开销较大，修改存储更甚。一个合约只能对它自己的存储进行读写。

第二个内存区被称为主存。合约执行每次消息调用时，都有一块新的，被清除过的主存。主存可以以字节粒度寻址，但是读写粒度为32字节（256比特）。操作主存的开销随着其增长而变大（平方级别）。

EVM不是基于寄存器，而是基于栈的虚拟机。因此所有的计算都在一个被称为栈的区域执行。栈最大有1024个元素，每个元素256比特。对栈的访问只限于其顶端，方式为：允许拷贝最顶端的16个元素中的一个到栈顶，或者是交换栈顶元素和下面16个元素中的一个。所有其他操作都只能取最顶的两个（或一个，或更多，取决于具体的操作）元素，并把结果压在栈顶。当然可以把栈上的元素放到存储或者主存中。但是无法只访问栈上指定深度的那个元素，在那之前必须要把指定深度之上的所有元素都从栈中移除才行。

指令集

EVM的指令集被刻意保持在最小规模，以尽可能避免可能导致共识问题的错误实现。所有的指令都是针对256比特这个基本的数据类型的操作。具备常用的算术，位，逻辑和比较操作。也可以做到条件和无条件跳转。此外，合约可以访问当前区块的相关属性，比如它的编号和时间戳。

消息调用

合约可以通过消息调用的方式来调用其它合约或者发送以太币到非合约账户。消息调用和交易非常类似，它们都有一个源，一个目标，数据负载，以太币，gas和返回数据。事实上每个交易都可以被认为是一个顶层消息调用，这个消息调用会依次产生更多的消息调用。

一个合约可以决定剩余gas的分配。比如内部消息调用时使用多少gas，或者期望保留多少gas。如果在内部消息调用时发生了out-of-gas异常（或者其他异常），合约将会得到通知，一个错误码被压在栈上。这种情况只是内部消息调用的gas耗尽。在solidity中，这种情况下发起调用的合约默认会触发一个人工异常。这个异常会打印出调用栈。就像之前说过的，被调用的合约（发起调用的合约也一样）会拥有崭新的主存并能够访问调用的负载。调用负载被存储在一个单独的被称为calldata的区域。调用执行结束后，返回数据将被存放在调用方预先分配好的一块内存中。

调用层数被限制为1024，因此对于更加复杂的操作，我们应该使用循环而不是递归。

代码调用和库

存在一种特殊类型的消息调用，被称为callcode。它跟消息调用几乎完全一样，只是加载自目标地址的代码将在发起调用的合约上下文中运行。

这意味着一个合约可以在运行时从另外一个地址动态加载代码。存储，当前地址和余额都指向发起调用的合约，只有代码是从被调用地址获取的。

这使得Solidity可以实现”库“。可复用的库代码可以应用在一个合约的存储上，可以用来实现复杂的数据结构。

日志

在区块层面，可以用一种特殊的可索引的数据结构来存储数据。这个特性被称为日志，Solidity用它来实现事件。合约创建之后就无法访问日志数据，但是这些数据可以从区块链外高效的访问。因为部分日志数据被存储在布隆过滤器（Bloom filter) 中，我们可以高效并且安全的搜索日志，所以那些没有下载整个区块链的网络节点（轻客户端）也可以找到这些日志。

创建

合约甚至可以通过一个特殊的指令来创建其他合约（不是简单的向零地址发起调用）。创建合约的调用跟普通的消息调用的区别在于，负载数据执行的结果被当作代码，调用者/创建者在栈上得到新合约的地址。

自毁

只有在某个地址上的合约执行自毁操作时，合约代码才会从区块链上移除。合约地址上剩余的以太币会发送给指定的目标，然后其存储和代码被移除。

注意，即使一个合约的代码不包含自毁指令，依然可以通过代码调用(callcode)来执行这个操作。

安装Solidity

基于浏览器的Solidity

如果你只是想尝试一个使用Solidity的小合约，你不需要安装任何东西，只要访问 基于浏览器的Solidity。

如果你想离线使用，你可以保存页面到本地，或者从 http://github.com/chriseth/browser-solidity 克隆一个。

NPM / node.js

这可能安装Solidity到本地最轻便最省事的方法。

在基于浏览器的Solidity上，Emscripten提供了一个跨平台JavaScript库，把C++源码编译为JavaScript，同时也提供NPM安装包。

去安装它就可以简单使用。，

```
npm install solc
```

如何使用nodejs包的详细信息可以在代码库中找到.

二进制安装包

Ethereum.

包括Mix IDE的二进制Solidity安装包在Ethereum网站C++ bundle中下载。

从源码构建

在MacOS X、Ubuntu和其它类Unix系统中编译安装Solidity非常相似。这个指南开始讲解如何在每个平台下安装相关的依赖软件，然后构建Solidity。

MacOS X

系统需求:

- OS X Yosemite (10.10.5)
- Homebrew
- Xcode

安装Homebrew：

```
  brew update
    brew install boost --c++11             # 这需要等待一段时间
    brew install cmake cryptopp miniupnpc leveldb gmp libmicrohttpd libjson-rpc-cpp
    # 仅仅安装Mix IDE和Alethzero
    brew install xz d-bus
    brew install llvm --HEAD --with-clang
    brew install qt5 --with-d-bus          # 如果长时间的等待让你发疯，那么添加--verbose输出
信息会让你感觉更好。
```

Ubuntu 系统

下面是在最新版Ubuntu系统上编译安装Solidity的指南。最佳的支持平台是2014年11月发布的64位Ubuntu 14.04，至少需要2GB内存。我们所有的测试都是基于此版本，当然我们也欢迎其它版本的测试贡献者。

安装依赖软件：

在你从源码编译之前，你需要准备一些工具和依赖软件。

首先，升级你的代码库。Ubuntu主代码库不提供所有的包，你需要从Ethereum PPA和LLVM获取。

注意

Ubuntu 14.04的用户需要使用：`sudo apt-add-repository ppa:george-edison55/cmake-3.x` 获取最新版本的cmake。

现在加入其它的包：

```
    sudo apt-get -y update
    sudo apt-get -y install language-pack-en-base
    sudo dpkg-reconfigure locales
    sudo apt-get -y install software-properties-common
    sudo add-apt-repository -y ppa:ethereum/ethereum
    sudo add-apt-repository -y ppa:ethereum/ethereum-dev
    sudo apt-get -y update
    sudo apt-get -y upgrade
```

对于Ubbuntu 15.04（Vivid Vervet）或者更老版本，使用下面的命令获取开发相关的包：

```
sudo apt-get -y install build-essential git cmake libboost-all-dev libgmp-dev liblevel
db-dev libminiupnpc-dev libreadline-dev libncurses5-dev libcurl4-openssl-dev libcrypto
pp-dev libjson-rpc-cpp-dev libmicrohttpd-dev libjsoncpp-dev libedit-dev libz-dev
```

对于Ubbuntu 15.10（Wily Werewolf）或者更新版本，使用下面的命令获取开发相关的包：

```
sudo apt-get -y install build-essential git cmake libboost-all-dev libgmp-dev liblevel
db-dev libminiupnpc-dev libreadline-dev libncurses5-dev libcurl4-openssl-dev libcrypto
pp-dev libjsonrpccpp-dev libmicrohttpd-dev libjsoncpp-dev libedit-dev libz-dev
```

不同版本使用不同获取命令的原因是，libjsonrpccpp-dev已经在最新版的Ubuntu的通用代码仓库中。

编译

如果你只准备安装solidity，忽略末尾Alethzero和Mix的错误。

```
    git clone --recursive https://github.com/ethereum/webthree-umbrella.git
    cd webthree-umbrella
    ./webthree-helpers/scripts/ethupdate.sh --no-push --simple-pull --project solidity

    # 更新Solidity库
    ./webthree-helpers/scripts/ethbuild.sh --no-git --project solidity --all --cores 4
 -DEVMJIT=0
    # 编译Solidity及其它
    # 在OS X系统加上DEVMJIT将不能编译，在Linux系统上则没问题
```

如果你选择安装Alethzero和Mix：

```
git clone --recursive https://github.com/ethereum/webthree-umbrella.git
    cd webthree-umbrella && mkdir -p build && cd build
    cmake ..
```

如果你想帮助Solidity的开发，你需要分支（fork）Solidity并添加到你的私人远端分支：

```
    cd webthree-umbrella/solidity
    git remote add personal git@github.com:username/solidity.git
```

注意webthree-umbrella使用的子模块,所以solidity是其自己的git代码库，但是他的设置不是保存在 `.git/config` , 而是在 `webthree-umbrella/.git/modules/solidity/config` .

Solidity 编程实例

Voting 投票

接下来的合约非常复杂，但展示了很多Solidity的特性。它实现了一个投票合约。当然，电子选举的主要问题是如何赋予投票权给准确的人，并防止操纵。我们不能解决所有的问题，但至少我们会展示如何委托投票可以同时做到投票统计是自动和完全透明。

思路是为每张选票创建一个合约，每个投票选项提供一个短名称。合约创建者作为会长将会给每个投票参与人各自的地址投票权。

地址后面的人们可以选择自己投票或者委托信任的代表人替他们投票。在投票结束后，winningProposal()将会返回获得票数最多的提案。

```
/// @title Voting with delegation.
/// @title 授权投票
   contract Ballot
   {
      // 这里声明了复杂类型
        // 将会在被后面的参数使用
          // 代表一个独立的投票人。
       struct Voter
       {
           uint weight; // 累积的权重。
           bool voted;  // 如果为真，则表示该投票人已经投票。
           address delegate; // 委托的投票代表
           uint vote;   // 投票选择的提案索引号
       }

      // 这是一个独立提案的类型
       struct Proposal
       {
           bytes32 name;   // 短名称（32字节）
           uint voteCount; // 累计获得的票数
       }
   address public chairperson;
 //这里声明一个状态变量，保存每个独立地址的`Voter` 结构
   mapping(address => Voter) public voters;
   //一个存储`Proposal`结构的动态数组
   Proposal[] public proposals;

   // 创建一个新的投票用于选出一个提案名`proposalNames`.
   function Ballot(bytes32[] proposalNames)
   {
       chairperson = msg.sender;
       voters[chairperson].weight = 1;

       //对提供的每一个提案名称，创建一个新的提案
       //对象添加到数组末尾
       for (uint i = 0; i < proposalNames.length; i++)
           //`Proposal({...})` 创建了一个临时的提案对象，
           //`proposal.push(...)`添加到了提案数组`proposals`末尾。
           proposals.push(Proposal({
               name: proposalNames[i],
               voteCount: 0
           }));
   }

    //给投票人`voter`参加投票的投票权，
```

```
    //只能由投票主持人`chairperson`调用。
    function giveRightToVote(address voter)
    {
        if (msg.sender != chairperson || voters[voter].voted)
            //`throw`会终止和撤销所有的状态和以太改变。
           //如果函数调用无效，这通常是一个好的选择。
           //但是需要注意，这会消耗提供的所有gas。
            throw;
        voters[voter].weight = 1;
    }

    // 委托你的投票权到一个投票代表 `to`。
    function delegate(address to)
    {
        // 指定引用
        Voter sender = voters[msg.sender];
        if (sender.voted)
            throw;

        //当投票代表`to`也委托给别人时，寻找到最终的投票代表
        while (voters[to].delegate != address(0) &&
               voters[to].delegate != msg.sender)
            to = voters[to].delegate;
        // 当最终投票代表等于调用者，是不被允许的。
        if (to == msg.sender)
            throw;
        //因为`sender`是一个引用，
        //这里实际修改了`voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter delegate = voters[to];
        if (delegate.voted)
            //如果委托的投票代表已经投票了，直接修改票数
            proposals[delegate.vote].voteCount += sender.weight;
        else
            //如果投票代表还没有投票，则修改其投票权重。
            delegate.weight += sender.weight;
    }

    ///投出你的选票（包括委托给你的选票）
    ///给 `proposals[proposal].name`。
    function vote(uint proposal)
    {
        Voter sender = voters[msg.sender];
        if (sender.voted) throw;
        sender.voted = true;
        sender.vote = proposal;
        //如果`proposal`索引超出了给定的提案数组范围
        //将会自动抛出异常，并撤销所有的改变。
        proposals[proposal].voteCount += sender.weight;
    }

   ///@dev 根据当前所有的投票计算出当前的胜出提案
    function winningProposal() constant
            returns (uint winningProposal)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++)
        {
            if (proposals[p].voteCount > winningVoteCount)
            {
                winningVoteCount = proposals[p].voteCount;
                winningProposal = p;
            }
        }
    }
}
```

可能的改进

现在，指派投票权到所有的投票参加者需要许多的交易。你能想出更好的方法么？

盲拍

这一节，我们将展示在以太上创建一个完整的盲拍合约是多么简单。我们从一个所有人都能看到出价的公开拍卖开始，接着扩展合约成为一个在拍卖结束以前不能看到实际出价的盲拍。

简单的公开拍卖

通常简单的公开拍卖合约，是每个人可以在拍卖期间发送他们的竞拍出价。为了实现绑定竞拍人的到他们的拍卖，竞拍包括发送金额/ether。如果产生了新的最高竞拍价，前一个最高价竞拍人将会拿回他的钱。在竞拍阶段结束后，受益人人需要手动调用合约收取他的钱 —— 合约不会激活自己。

```
contract SimpleAuction {
    // 拍卖的参数。
   // 时间要么为unix绝对时间戳（自1970-01-01以来的秒数），
   // 或者是以秒为单位的出块时间
    address public beneficiary;
    uint public auctionStart;
    uint public biddingTime;

    //当前的拍卖状态
    address public highestBidder;
    uint public highestBid;

   //在结束时设置为true来拒绝任何改变
    bool ended;

   //当改变时将会触发的Event
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    //下面是一个叫做natspec的特殊注释，
    //由3个连续的斜杠标记，当询问用户确认交易事务时将显示。

    ///创建一个简单的合约使用`_biddingTime`表示的竞拍时间，
   /// 地址`_beneficiary`.代表实际的拍卖者
    function SimpleAuction(uint _biddingTime,
                           address _beneficiary) {
        beneficiary = _beneficiary;
        auctionStart = now;
        biddingTime = _biddingTime;
    }

    ///对拍卖的竞拍保证金会随着交易事务一起发送，
   ///只有在竞拍失败的时候才会退回
    function bid() {

       //不需要任何参数，所有的信息已经是交易事务的一部分
        if (now > auctionStart + biddingTime)
           //当竞拍结束时撤销此调用
            throw;
        if (msg.value <= highestBid)
           //如果出价不是最高的，发回竞拍保证金。
```

```
            throw;
        if (highestBidder != 0)
            highestBidder.send(highestBid);
        highestBidder = msg.sender;
        highestBid = msg.value;
        HighestBidIncreased(msg.sender, msg.value);
    }

   ///拍卖结束后发送最高的竞价到拍卖人
    function auctionEnd() {
        if (now <= auctionStart + biddingTime)
            throw;
            //拍卖还没有结束
        if (ended)
            throw;
     //这个收款函数已经被调用了
        AuctionEnded(highestBidder, highestBid);
        //发送合约拥有所有的钱，因为有一些保证金可能退回失败了。

        beneficiary.send(this.balance);
        ended = true;
    }

    function () {
        //这个函数将会在发送到合约的交易事务包含无效数据
        //或无数据的时执行，这里撤销所有的发送，
        //所以没有人会在使用合约时因为意外而丢钱。
        throw;
    }
}
```

Blind Auction 盲拍

接下来扩展前面的公开拍卖成为一个盲拍。盲拍的特点是拍卖结束以前没有时间压力。在一个透明的计算平台上创建盲拍系统听起来可能有些矛盾，但是加密算法能让你脱离困境。

在拍卖阶段, 竞拍人不需要发送实际的出价，仅仅只需要发送一个它的散列值。因为目前几乎不可能找到两个值（足够长）的散列值相等，竞拍者提交他们的出价散列值。在拍卖结束后，竞拍人重新发送未加密的竞拍出价，合约将检查其散列值是否和拍卖阶段发送的一样。另一个挑战是如何让拍卖同时实现绑定和致盲 ：防止竞拍人竞拍成功后不付钱的唯一的办法是，在竞拍出价的同时发送保证金。但是在Ethereum上发送保证金是无法致盲，所有人都能看到保证金。下面的合约通过接受任何尽量大的出价来解决这个问题。当然这可以在最后的揭拍阶段进行复核，一些竞拍出价可能是无效的，这样做的目的是（它提供一个显式的标志指出是无效的竞拍，同时包含高额保证金）：竞拍人可以通过放置几个无效的高价和低价竞拍来混淆竞争对手。

```
contract BlindAuction
{
    struct Bid
    {
        bytes32 blindedBid;
        uint deposit;
    }
    address public beneficiary;
    uint public auctionStart;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;
```

```
    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    event AuctionEnded(address winner, uint highestBid);

   ///修饰器（Modifier）是一个简便的途径用来验证函数输入的有效性。
   ///`onlyBefore` 应用于下面的 `bid`函数，其旧的函数体替换修饰器主体中 `_`后就是其新的函数体
    modifier onlyBefore(uint _time) { if (now >= _time) throw; _ }
    modifier onlyAfter(uint _time) { if (now <= _time) throw; _ }

    function BlindAuction(uint _biddingTime,
                            uint _revealTime,
                            address _beneficiary)
    {
        beneficiary = _beneficiary;
        auctionStart = now;
        biddingEnd = now + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }

   ///放置一个盲拍出价使用`_blindedBid`=sha3(value,fake,secret).
   ///仅仅在竞拍结束正常揭拍后退还发送的以太。当随同发送的以太至少
   ///等于 "value"指定的保证金并且 "fake"不为true的时候才是有效的竞拍
   ///出价。设置 "fake"为true或发送不合适的金额将会掩没真正的竞拍出
   ///价，但是仍然需要抵押保证金。同一个地址可以放置多个竞拍。
    function bid(bytes32 _blindedBid)
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: _blindedBid,
            deposit: msg.value
        }));
    }

   ///揭开你的盲拍竞价。你将会拿回除了最高出价外的所有竞拍保证金
   ///以及正常的无效盲拍保证金。
    function reveal(uint[] _values, bool[] _fake,
                    bytes32[] _secret)
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        if (_values.length != length || _fake.length != length ||
                    _secret.length != length)
            throw;
        uint refund;
        for (uint i = 0; i < length; i++)
        {
            var bid = bids[msg.sender][i];
            var (value, fake, secret) =
                    (_values[i], _fake[i], _secret[i]);
            if (bid.blindedBid != sha3(value, fake, secret))
                //出价未被正常揭拍，不能取回保证金。
                continue;
            refund += bid.deposit;
            if (!fake && bid.deposit >= value)
                if (placeBid(msg.sender, value))
                    refund -= value;
            //保证发送者绝不可能重复取回保证金
            bid.blindedBid = 0;
        }
        msg.sender.send(refund);
    }

    //这是一个内部 (internal)函数，
   //意味着仅仅只有合约（或者从其继承的合约）可以调用
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
```

```
        if (value <= highestBid)
            return false;
        if (highestBidder != 0)
            //退还前一个最高竞拍出价
            highestBidder.send(highestBid);
        highestBid = value;
        highestBidder = bidder;
        return true;
    }

   ///竞拍结束后发送最高出价到竞拍人
    function auctionEnd()
        onlyAfter(revealEnd)
    {
        if (ended) throw;
        AuctionEnded(highestBidder, highestBid);
        //发送合约拥有所有的钱，因为有一些保证金退回可能失败了。
        beneficiary.send(this.balance);
        ended = true;
    }

    function () { throw; }
}
Safe Remote Purchase 安全的远程购物

contract Purchase
{
    uint public value;
    address public seller;
    address public buyer;
    enum State { Created, Locked, Inactive }
    State public state;
    function Purchase()
    {
        seller = msg.sender;
        value = msg.value / 2;
        if (2 * value != msg.value) throw;
    }
    modifier require(bool _condition)
    {
        if (!_condition) throw;
        _
    }
    modifier onlyBuyer()
    {
        if (msg.sender != buyer) throw;
        _
    }
    modifier onlySeller()
    {
        if (msg.sender != seller) throw;
        _
    }
    modifier inState(State _state)
    {
        if (state != _state) throw;
        _
    }
    event aborted();
    event purchaseConfirmed();
    event itemReceived();

   ///终止购物并收回以太。仅仅可以在合约未锁定时被卖家调用。
    function abort()
        onlySeller
        inState(State.Created)
    {
        aborted();
        seller.send(this.balance);
        state = State.Inactive;
    }
```

```
    ///买家确认购买。交易包含两倍价值的（`2 * value`）以太。
    ///这些以太会一直锁定到收货确认(confirmReceived)被调用。
     function confirmPurchase()
         inState(State.Created)
         require(msg.value == 2 * value)
     {
         purchaseConfirmed();
         buyer = msg.sender;
         state = State.Locked;
     }

     ///确认你（买家）收到了货物，这将释放锁定的以太。
     function confirmReceived()
         onlyBuyer
         inState(State.Locked)
     {
         itemReceived();
         buyer.send(value);//我们有意忽略了返回值。
         seller.send(this.balance);
         state = State.Inactive;
     }
     function() { throw; }
 }
```

小额支付通道

待补

深入理解 Solidity

此节将帮助你深入理解Solidity，如果有遗漏，请和我们联系Gitter或者在Githhub上发pull request

Layout of a Solidity Source File

- Importing other Source Files
- Comments
- Solidity源文件的布局
- 引入其他的源文件
- 注释
- Structure of a Contract
- 合约的结构
- Types
- Value Types
- Enums
- Reference Types
- Mappings
- Operators Involving LValues
- Conversions between Elementary Types
- Type Deduction

类型

- 变量类型
- 枚举
- 参考类型
- 映射
- 包括左赋值的操作符
- 在基本类型间的转换

- 类型导出
- Units and Globally Available Variables
- Ether Units
- Time Units
- Special Variables and Functions
- 单元局部和全局可见变量
- Ether 单元
- Time 单元
- 特殊变量和函数
- Expressions and Control Structures
- Control Structures
- Function Calls
- Order of Evaluation of Expressions
- Assignment
- Exceptions
- 表达式和控制结构
- 控制结构
- 函数调用
- 表达式计算顺序
- 赋值
- 异常
- Contracts
- Creating Contracts
- Visibility and Accessors
- Function Modifiers
- Constants
- Fallback Function

- Events
- Inheritance
- Abstract Contracts
- Libraries
- Using For
- 合约
- 创立合约
- 可见性和访问性
- 函数修饰符
- 常量
- 回退功能
- 事件
- 继承
- 抽象合约
- 库
- 用作
- Miscellaneous
- Layout of State Variables in Storage
- Esoteric Features
- Internals - the Optimizer
- Using the Commandline Compiler
- Tips and Tricks
- Pitfalls
- Cheatsheet
- 杂项
- 存储器中状态变量的布局
- 深奥的特性

- 内部-优化器
- 用命令行编译器
- 提示和技巧
- “坑”
- 备忘录

源文件的布局

源文件包括任意数量的合约定义和include指令

引入其他源文件

语法和语义

Solidity支持 import语句，非常类似于JavaScript（ES6）,虽然Solidity不知道“缺省导出”的概念。

在全局层次上，你可以用下列形式使用import语句

```
import "filename";
```

将会从"filename"导入所有的全局符号(和当前导入的符号)到当前的全局范围里（不同于ES6，但是Solidity保持向后兼容）

```
**import** ***** as symbolName from "filename";
```

创立了一个全局的符号名 **symbolName**,其中的成员就来自“filename”的所有符号

```
import {symbol1 as alias, symbol2} from “filename”;
```

将创立一个新的全局变量别名：alias 和 symbol2， 它将分别从"filename" 引入symbol1 和 symbol2

另外，Solidity语法不是ES6的子集，但可能（使用）更便利

```
import “filename” as symbolName;
```

等价于 import * as symbolName from “filename”;.

路径

- 在上面，文件名总是用/作为目录分割符，. 是当前的目录，.. 是父目录，路径名称不用. 开头的都将视为绝对路径。
- 从同一个目录下import 一个文件 x 作为当前文件，用 import ”./x” as x; 如果使用

- import “x” as x; 是不同的文件引用（在全局中使用"include directory"）,。
- 它将依赖于编译器（见后）来解析路径。通常，目录层次不必严格限定映射到你的本地文件系统，它也可以映射到ipfs,http或git上的其他资源

使用真正的编译器

当编译器启动时，不仅可以定义如何找到第一个元素的路径，也可能定义前缀重映射的路径，如 github.com/ethereum/dapp-bin/library将重映射到 /usr/local/dapp-bin/library ，编译器将从这个路径下读取文件。 如果重映射的keys是前缀，（编译器将尝试）最长的路径。允许回退并且映射到“/usr/local/include/solidity”。

solc:

solc(行命令编译器)，重映射将提供 key=值参数, =值 部分是可选的（缺省就是key ） 。

所有重映射的常规文件都将被编译（包括他们的依赖文件），这个机制完全向后兼容（只要没有文件名包含 a=） ，这不是一个很大的变化，

```
比如 ，如果你从 github.com/ethereum/dapp-bin/ 克隆到本地 /usr/local/dapp-bin, 你可以用下列源文件
```

import “github.com/ethereum/dapp-bin/library/iterable_mapping.sol” as it_mapping;

运行编译器时如下

solc github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ source.sol

注意： solc仅仅允许你从特定的目录下include文件，他们必须是一个显式定义的，包含目录或子目录的源文件， 或者是重映射目标的目录（子目录）。如果你允许直接include, 要增加remapping =/.

如果有多个重映射，就要做一个合法文件，文件中选择最长的公共前缀

基于浏览器的solidity

基于浏览器的编译器提供了从github上的自动重映射,并且自动检索网络上的文件，你可以import 迭代映射 如

```
import “github.com/ethereum/dapp-bin/library/iterable_mapping.sol” as it_mapping;.
```

其他源代码提供者可以以后增加进来。

注释

可用单行注释 (//) 和多行注释 (/…/)

有种特别的注释 叫做 “natspec ” （文档以后写出来），在函数声明或定义的右边用三个斜杠（///）或者用 两个星号 (/ **... */).** 。如果想要调用一个函数，可以使用doxygen-style标签里面文档功能,形式验证,并提供一个确认条件的文本注释显示给用户 。**

合约的结构

Solidity的合约和面向对象语言中的类的定义相似。每个合约包括了 状态变量，函数，函数修饰符，事件，结构类型 和枚举类型。另外，合约也可以从其他合约中继承 。

- 状态变量是在合约存贮器中永久存贮的值
- 函数是合约中可执行单位的代码
- 函数修饰符可以在声明的方式中补充函数的语义
- 事件是和EVM（以太虚拟机）日志设施的方便的接口
- 结构是一组用户定义的变量
- 枚举是用来创建一个特定值的集合的类型

类型

Solidity是一种静态类型语言，意思是每个变量（声明和本地）在编译时刻都要定义 （或者至少要知晓，参看 后面的类型导出 ）。

Solidity提供几个基本类型组合成复杂类型。

变量类型

以下类型被叫做值类型，因为这些类型的变量总是要被赋值，作为函数参数或者在赋值中，总需要拷贝。

布尔类型

布尔：可能的常量值 是 真或假

操作符：

- ！（逻辑非）
- &&（逻辑与，“and"）
- || (逻辑或，”or“)
- ==（相等）
- ！=（不等）
- 操作符||和&&可以应用常规短路规则，即 表达式 f(x) || g(y), 如果f(x) 已是真，g(y）将不用计算，即使它有副作用 （真||任意值 均为真，假&&任意布尔值 均为假）。

整型

int• / uint•: 是有符号和无符号的整数，关键字uint8 到 uint256 步长8 (从8到256位的无符号整数)

uint 和 int 分别是 uint256 和 int256的别名

操作符：

- 比较 ： <=,<,==,!=,>=,> (计算布尔量)
- 位操作符： &，|，^(位异或)，~（位取反）

- 算术操作符：+，-，一元-，一元+，，/,%(取余数)，*（幂次方）

地址

地址： 20字节（一个 Ethereum地址），地址的类型也可以有成员（请看地址功能(#functions-on-addresses)） 作为所有合约的base

操作符：

- <=, <, ==, !=, >= 和 >

地址成员：

- 账户余额（balance）和发送（send）

若查询到有资产余额的地址，然后发送 Ether(以wei为单位) 到send 函数的地址上

```
address x = 0x123;

address myAddress = this;

if (x.balance < 10 && myAddress.balance >= 10) x.send(10);
```

注解

如果x是合约地址，它的代码（特别的指出 如果有回退函数，） 将和send 调用一起执行（这是EVM的限制，不能修改） 如果用完了gas或者失败，Ether转移将被回退，这种情况下，send返回false

- 调用(call)和调用码(callcode)

另外，和合约的接口不是附在ABI 上，函数调用可以引用任意数量的参数，这些参数要填补成32字节，并被拼接。一个例外的情况是第一个参数被确切的编码成4字节，这种情况下，不用填补，直接使用函数符号

```
address nameReg = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;

nameReg.call("register", "MyName");

nameReg.call(bytes4(sha3("fun(uint256)")), a);
```

函数调用返回了一个布尔值，表示函数是否是正常调用结束(true)或引起了EVM异常(false)。不可能访问返回实际数据(这个我们需要提前知道编码和大小)。

同样,可以使用函数callcode:不同之处在于,只使用给定地址的编码,所有其他方面(存储、余额…)取自于当前的合约。callcode的目的是使用库代码存储在另一个合同。用户必须确保存储在两个合约的布局适用于callcode。

call和callcode是非常低级的函数，它可以作为打破Solidity的类型安全的最后手段。

- call and callcode

请注意

所有合同继承成员地址,所以可以使用this.balance查询当前合约的余额。

```
Fixed-size byte arrays

bytes1, bytes2, bytes3, ..., bytes32. byte is an alias for bytes1.

Operators:
```

- Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)
- Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)

固定大小的字节数组

bytes1, bytes2, bytes3, ..., bytes32. byte 都是 bytes1的别名.

操作符：

- 比较符：<=, <, ==, !=, >=, > (布尔的评估)
- 位操作符;: &, |, ^ (按位置异或)，~ （按位取反）

动态分配大小字节数组:

bytes:动态分配大小字节数组,参看 Arrays,不是一个值类型!

string:动态大小UTF8编码的字符串,参看Arrays。不是一个值类型!

```
  作为一个常识, 使用bytes来表示任意长度原始字节数据，使用string来表示特定长度字符串数据(utf - 8编码
)。如果你想限定特定数量的字节长度, 就使用bytes1 到 bytes32，  因为这样占用的存储空间更少。
```

整型常量

整型常量是特定精度整数，它们也可以和非常量同时使用。例如, var x = 1 - 2; 1 - 2的值是-1, 然后赋值给x, 这时x接收类型为int8——最小的类型,其中包含-1, 虽然1和2的类型实际上是uint8。

有时最大超过256位的整型常量也可用于计算:var x =(0 xffffffffffffffffffff *0 xffffffffffffffffffff*) 0;这里,x的值是0，它的类型是uint8类型。

字符串常量

字符串常量用两个双引号括起来(“abc”)。和整型常量相比,字符串常量有些不同,字符串常量可以隐式转换成bytes • 如果合适,可以是bytes,也可以是string。

```
contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;
    function setGoStraight()
    {
        choice = ActionChoices.GoStraight;
    }
```

枚举是一种Solidity中的创建一个用户定义的类型。枚举类型中的枚举值可显式转换，但从整数类型隐式转换是不允许的。

```
contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;
    function setGoStraight()
    {
        choice = ActionChoices.GoStraight;
    }
    // Since enum types are not part of the ABI, the signature of "getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity. The integer type used is just
    // large enough to hold all enum values, i.e. if you have more values,
    // `uint16` will be used and so on.
/ /因为枚举类型不是ABI的一部分,“getChoice”的符号
/ /将自动改为“getChoice()返回(uint8)”
/ /从Solidity外部看，使用的整数类型
/ /足够容纳所有枚举值,但如果你有更多的值,
/ /“uint16”将使用。
    function getChoice() returns (ActionChoices)
    {
        return choice;
    }
    function getDefaultChoice() returns (uint)
    {
        return uint(defaultChoice);
    }}
```

引用类型

复杂类型,例如类型并不总是适合256位，比我们已经看到的值类型更复杂的类型，必须更仔细地处理。因为复制拷贝他们可能相当耗费存储和时间， 我们必须考虑把它们存储在内存(这不是持久化)或者存储器(状态变量存放的地方)。

数据位置

每一个复杂类型,即数组和结构体,有一个额外的注解,“数据位置”,不管它是存储在内存中，还是存储在存储器上。根据上下文,总是有一个默认的,但它可以通过附加存储或内存覆盖类型。函数参数的默认值(包括返回参数)是在内存上,局部变量的默认存储位置是在存储器上。存储器上存有状态变量(很明显)。

（除了内存，存储器这两个位置之外），还有第三个数据位置,“calldata”,这是一个 无法改变的，非持久的 存储函数参数的地方。外部函数的函数参数(不返回参数)“calldata”，其在形式上象内存。

数据位置很重要,因为它们改变赋值方式:在存储和内存以及状态变量之间赋值(甚至从其他状态变量)总是创建一个独立的副本。赋值只分配一个本地存储变量引用,这总是指向状态变量的引用,后者同时改变。另一方面,从一个内存存储引用类型, 赋值到另一个内存存储引用类型，（这时）并不创建一个副本。

```
contract c {
  uint[] x; // the data location of x is storage
  // the data location of memoryArray is memory
  function f(uint[] memoryArray) {
    x = memoryArray; // works, copies the whole array to storage
    var y = x; // works, assigns a pointer, data location of y is storage
    y[7]; // fine, returns the 8th element
    y.length = 2; // fine, modifies x through y
    delete x; // fine, clears the array, also modifies y
    // The following does not work; it would need to create a new temporary /
    // unnamed array in storage, but storage is "statically" allocated:
    // y = memoryArray;
    // This does not work either, since it would "reset" the pointer, but there
    // is no sensible location it could point to.
    // delete y;
    g(x); // calls g, handing over a reference to x
    h(x); // calls h and creates an independent, temporary copy in memory
  }
  function g(uint[] storage storageArray) internal {}
  function h(uint[] memoryArray) {}}
contract c {
  uint[] x; // the data location of x is storage    x的数据位置是存储器
  // the data location of memoryArray is memory  memoryArray的数据位置是内存
  function f(uint[] memoryArray) {
    x = memoryArray; // works, copies the whole array to storage  运行，拷贝整个数组到存储
```

```
器

    var y = x; // works, assigns a pointer, data location of y is storage 运行，赋值到一个指针，y的数据位置是存储器

    y[7]; // fine, returns the 8th element 好了，返回第8个元素

    y.length = 2; // fine, modifies x through y    好了，通过y改变x

    delete x; // fine, clears the array, also modifies y  好了，清除数组，也改变y

    // The following does not work; it would need to create a new temporary /  以下代码不起作用，  它是在存储中创立一个临时的未命名的数组，但存储器是“静态”分配的

    // unnamed array in storage, but storage is "statically" allocated:

    // y = memoryArray;

    // This does not work either, since it would "reset" the pointer, but there 这个也不起作用，因为 它重置了指针， 但已经没有相应的位置可以指向

    // is no sensible location it could point to.

    // delete y;

    g(x); // calls g, handing over a reference to x   调用g(x)  将x作为引用

    h(x); // calls h and creates an independent, temporary copy in memory 调用h(x). 在内存中创立了一个独立的，暂时的拷贝

  }

  function g(uint[] storage storageArray) internal {}

  function h(uint[] memoryArray) {}}
```

总结

强制数据位置:

- 外部函数的参数(不返回):calldata
- 状态变量:存储器

默认数据位置:

- 函数（有返回）的参数:内存
- 其他所有局部变量:存储器

数组

数组是可以在编译时固定大小的，也可以是动态的。对于存储器数组来说，成员类型可以是任意的(也可以是其他数组，映射或结构)。对于内存数组来说 ,成员类型不能是一个映射；如果是公开可见的函数参数，成员类型是必须是ABI类型的。

固定大小k的数组和基本类型T，可以写成T[k], 动态数组写成 T[ ] 。例如, 有5个基本类型为uint 的动态数组的数组 可以写成uint[ ][5] (注意,和一些其他语言相比，这里的符号表示次序是反过来的)。为了访问第三动态数组中的第二个uint, 必须使用x[2]1。

*bytes*和 string 是特殊类型的数组。 bytes 类似于byte[ ],但它是紧凑排列在calldata里的。string 等于 bytes ， 但不允许用长度或所以索引访问(现在情况是这样的)。

所以bytes应该优先于byte[ ],因为它效率更高。

请注意

如果你想访问字符串s的某个字节, 要使用 bytes(s).length/bytes(s)[7]= ' x ';。记住,你正在访问的低级utf - 8字节表示,而不是单个字符!

成员（函数）：

length: 总有 一个称作length的成员（函数）来存放元素的数量。动态数组可以通过改变.length成员（函数），在存储器里来调整大小（不是在内存中）。当试图访问现有长度之外的成员时，这并不是自动被许可的。（数组）一旦创建，内存里的数组大小是固定的(如果是动态的数组,则取决于运行时参数)。

push :动态存储数组arrays和字节bytes(不是字符串string)有一个成员函数称作push,可在数组的尾部添加一个元素。函数返回新的长度。

警告

到目前为止，还不可以在外部函数中使用数组的数组。

警告

由于EVM的局限,不可能从外部函数调用返回的动态内容。合约函数f contract C { function f() returns (uint[]) { ... } } 使用web3.js调用,将有返回值, 但使用Solidity调用，就没有返回值。

现在唯一的解决方法是使用较大的静态尺寸大小的数组。

```
contract ArrayContract {

  uint[2*\*20] m_aLotOfIntegers;

  // Note that the following is not a pair of arrays but an array of pairs.

  bool[2][] m_pairsOfFlags;

  // newPairs is stored in memory - the default for function arguments

  function setAllFlagPairs(bool[2][] newPairs) {

    // assignment to a storage array replaces the complete array

    m_pairsOfFlags = newPairs;

  }

  function setFlagPair(uint index, bool flagA, bool flagB) {
```

```
    // access to a non-existing index will throw an exception
    m_pairsOfFlags[index][0] = flagA;
    m_pairsOfFlags[index][1] = flagB;
  }
  function changeFlagArraySize(uint newSize) {
    // if the new size is smaller, removed array elements will be cleared
    m_pairsOfFlags.length = newSize;
  }
  function clear() {
    // these clear the arrays completely
    delete m_pairsOfFlags;
    delete m_aLotOfIntegers;
    // identical effect here
    m_pairsOfFlags.length = 0;
  }
  bytes m_byteData;
  function byteArrays(bytes data) {
    // byte arrays ("bytes") are different as they are stored without padding,
    // but can be treated identical to "uint8[]"
    m_byteData = data;
    m_byteData.length += 7;
    m_byteData[3] = 8;
    delete m_byteData[2];
  }
  function addFlag(bool[2] flag) returns (uint) {
    return m_pairsOfFlags.push(flag);
  }
  function createMemoryArray(uint size) returns (bytes) {
    // Dynamic memory arrays are created using `new`:
    uint[2][] memory arrayOfPairs = new uint[2][](size);
    // Create a dynamic byte array:
    bytes memory b = new bytes(200);
    for (uint i = 0; i < b.length; i++)
      b[i] = byte(i);
    return b;
  }}
```

```
contract ArrayContract {

  uint[2\20] m_aLotOfIntegers;

  // Note that the following is not a pair of arrays but an array of pairs. 注意下面不是两个数组，而是一个数组，该数组的成员是一对值

  bool[2][] m_pairsOfFlags;

  // newPairs is stored in memory - the default for function arguments  newPairs在内存中存储-这是函数参数的缺省方式

  function setAllFlagPairs(bool[2][] newPairs) {

    // assignment to a storage array replaces the complete array 赋值到一个存储器数组里以替换整个数组

    m_pairsOfFlags = newPairs;

  }

  function setFlagPair(uint index, bool flagA, bool flagB) {

    // access to a non-existing index will throw an exception

    m_pairsOfFlags[index][0] = flagA;

    m_pairsOfFlags[index][1] = flagB;

  }

  function changeFlagArraySize(uint newSize) {

    // if the new size is smaller, removed array elements will be cleared  如果新的尺寸太小，则已经移除的元素将被清除

    m_pairsOfFlags.length = newSize;

  }

  function clear() {

    // these clear the arrays completely

    delete m_pairsOfFlags;

    delete m_aLotOfIntegers;

    // identical effect here

    m_pairsOfFlags.length = 0;

  }

  bytes m_byteData;

  function byteArrays(bytes data) {

    // byte arrays ("bytes") are different as they are stored without padding,  如果没有填充的话，字节数组（"bytes"）和存储时是不同的

    // but can be treated identical to "uint8[]"  但可以转换成 "uint8[]"

    m_byteData = data;

    m_byteData.length += 7;

    m_byteData[3] = 8;

    delete m_byteData[2];

  }
```

```
  function addFlag(bool[2] flag) returns (uint) {
    return m_pairsOfFlags.push(flag);
  }
  function createMemoryArray(uint size) returns (bytes) {
    // Dynamic memory arrays are created using `new`: 使用`new`创立动态内存数组
    uint[2][] memory arrayOfPairs = new uint[2][](size);
    // Create a dynamic byte array:  创立动态 byte 数组
    bytes memory b = new bytes(200);
    for (uint i = 0; i < b.length; i++)
      b[i] = byte(i);
    return b;
  }}
```

结构体

Solidity 提供了一种方法来定义新类型的形式结构,如下面的例子所示:

```
contract CrowdFunding {
  // Defines a new type with two fields.
  struct Funder {
    address addr;
    uint amount;
  }
  struct Campaign {
    address beneficiary;
    uint fundingGoal;
    uint numFunders;
    uint amount;
    mapping (uint => Funder) funders;
  }
  uint numCampaigns;
  mapping (uint => Campaign) campaigns;
  function newCampaign(address beneficiary, uint goal) returns (uint campaignID) {
    campaignID = numCampaigns++; // campaignID is return variable
    // Creates new struct and saves in storage. We leave out the mapping type.
    campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
```

```
  }
  function contribute(uint campaignID) {
    Campaign c = campaigns[campaignID];
        // Creates a new temporary memory struct, initialised with the given values
        // and copies it over to storage.
        // Note that you can also use Funder(msg.sender, msg.value) to initialise.
    c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
    c.amount += msg.value;
  }
  function checkGoalReached(uint campaignID) returns (bool reached) {
    Campaign c = campaigns[campaignID];
    if (c.amount < c.fundingGoal)
      return false;
    c.beneficiary.send(c.amount);
    c.amount = 0;
    return true;
  }}
contract CrowdFunding {
  // Defines a new type with two fields. 定义了两个域的新类型
  struct Funder {
    address addr;
    uint amount;
  }
  struct Campaign {
    address beneficiary;
    uint fundingGoal;
    uint numFunders;
    uint amount;
    mapping (uint => Funder) funders;
  }
  uint numCampaigns;
  mapping (uint => Campaign) campaigns;
  function newCampaign(address beneficiary, uint goal) returns (uint campaignID) {
    campaignID = numCampaigns++; // campaignID is return variable  campaignID是返回的变量
    // Creates new struct and saves in storage. We leave out the mapping type. 创建一个新的结构体，保存在存储器里， 保留了映射类型
```

```
    campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);

  }

  function contribute(uint campaignID) {

    Campaign c = campaigns[campaignID];

        // Creates a new temporary memory struct, initialised with the given values 创建了一个新的临时内存结构体，用给定的值进行初始化

        // and copies it over to storage. 拷贝到存储器上

        // Note that you can also use Funder(msg.sender, msg.value) to initialise. 注意你可以使用 Funder(msg.sender, msg.value)来初始化

    c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});

    c.amount += msg.value;

  }

  function checkGoalReached(uint campaignID) returns (bool reached) {

    Campaign c = campaigns[campaignID];

    if (c.amount < c.fundingGoal)

      return false;

    c.beneficiary.send(c.amount);

    c.amount = 0;

    return true;

  }}
```

此（例子）合约没有提供众筹合约的完整功能, 但它包含了必要的基本概念，以便（让我们更好地）理解结构体。结构体类型可以是内部映射或者是数组，他们本身也可以包含映射和数组。

通常这是不可能,即一个结构体包含一个自身类型的成员, 虽然结构体本身可以是一个映射的值类型成员。这个限制是必要的, 原因是结构体的大小是有限的。

注意所有的函数中, 结构类型是赋值给一个局部变量(默认存储数据的位置)。这并不复制结构体，仅仅保存了一个引用， 本地变量的赋值最终还是以写进了状态中。

当然,您也可以直接访问结构体的成员变量，而不用赋值到一个局部变量,如 campaigns[campaignID].amount = 0.

映射

映射类型被声明为 mapping _KeyType => _ValueType, _KeyType可以是除了映射以外的其他任何类型，_ValueType可以是任何类型,包括映射。

映射可以被视为初始化的散列表,这样每一个键值都存在, 这些键值在字节表示上是全零。相似性到此为止,尽管:key数据实际上并不是存储在一个映射中,它只有在使用sha3哈希查找值使用。

因此,映射没有长度，也没有一个键或值的被“set”的概念。

映射是只允许为状态变量(在内部函数中作为一个存储引用类型)。

包括左值操作的操作符

如果是一个左值操作(即一个可以赋值给它的变量),可以使用以下的操作符:

a += e相当于 a = a + e。操作符- = * =,/ = % = | = & = ^ = 都有相应的定义。a++和a--相当于 a+ = 1 /a - = 1,但是表达式本身还有一个操作前的值。相比之下, --a和++a有相同的影响但返回值改变。

删除

删除一个指定类型的初始值为整数,即相当于a= 0,但是它也可以用于数组,它分配一个动态数组的长度为零或一个静态数组长度相同的所有元素重置。对于结构体,它分配一个struct,重置所有成员。

删除没有影响整体映射(如映射的键可能是任意的,通常是未知的)。如果你删除一个结构,它将重置没有映射的所有成员，也可以是递归的成员,除非它们映射。然而,个别键和他们的映射是可以删除。

重要的是要注意,删除一个a的赋值, 即它存储在一个新的对象。

```
contract DeleteExample {

  uint data;

  uint[] dataArray;

  function f() {

    uint x = data;

    delete x; // sets x to 0, does not affect data

    delete data; // sets data to 0, does not affect x which still holds a copy

    uint[] y = dataArray;

    delete dataArray; // this sets dataArray.length to zero, but as uint[] is a comple
x object, also

    // y is affected which is an alias to the storage object

    // On the other hand: "delete y" is not valid, as assignments to local variables

    // referencing storage objects can only be made from existing storage objects.

  }}
contract DeleteExample {

  uint data;

  uint[] dataArray;

  function f() {

    uint x = data;

    delete x; // sets x to 0, does not affect data  设置x为0, 不影响data

    delete data; // sets data to 0, does not affect x which still holds a copy  设置dat
a为0,x不受影响，x仍然有一个拷贝

    uint[] y = dataArray;

    delete dataArray; // this sets dataArray.length to zero, but as uint[] is a comple
x object, also dataArray.length长度是0。但是uint[ ]是一个复杂对象， y受影响，其是存储对象的别名

    // y is affected which is an alias to the storage object

    // On the other hand: "delete y" is not valid, as assignments to local variables 另
外， "delete y"是非法的，因为y是赋值到本地变量

    // referencing storage objects can only be made from existing storage objects.引用
存储对象仅仅来自于现有的存储对象

  }}
```

基本类型之间的转换

隐式转换

如果一个操作符应用于不同类型，编译器(就会)试图隐式把操作数的类型，从一种类型转换到其他类型(赋值也是如此)。一般来说,一个隐式的值类型之间的转换是可能的,如果是语义敏感的话，信息不会丢失:unt8可转换成uint16， int128， int256, 但int8不能转换成uint256(因为uint256放不下 如 -1)。此外,无符号整数可以转换成相同或更大的尺寸的bytes , 但反过来，不行 。任何类型都可以转化为uint160，也可以转换为地址。

显式转换

如果编译器不允许隐式转换，但你知道你在做什么,一个显式的类型转换有时是可能的:

```
int8 y = 3;

uint x =uint(y);
```

这个代码片段结尾 ， x的值是 0xfffff . .fd(64个十六进制字符),-3在256位的二进制补码表示。

如果一个类型是显式地转换为一个更小的类型,高阶位将被移除。

```
uint32 = 0x12345678;

uint16 b = uint16(a);/ / *b will be 0x5678 now*  *b现在变成了0x5678，（少了1234）*
```

类型推导

为方便起见,它并不总是必须显式地指定一个变量的类型,编译器会自动从第一个赋值表达式的变量类型里推断出新变量的类型:

```
uint20 x = 0 x123;

var y = x;
```

在这里,y的类型是uint20。在函数参数或返回参数是不可能使用var（这个关键字）的。

警告

这个类型仅仅是从第一次赋值推导得出的,所以以下代码片段的循环是无限的, 因为 i 的类型是uint8， 这种类型的任何值都小于2000。for (var i = 0;< 2000;i+ +){…}

Next Previous

单位和全局可用变量

以太单位

数词后面可以有一个后缀, wei, finney, szabo 或 ether 和 ether 相关量词 之间的转换,在以太币数量后若没有跟后缀，则缺省单位是“wei“, 如 2 ether == 2000 finney （这个表达式）计算结果为true。

时间单位

后缀的秒,分,小时,天,周,年， 数量词的时间单位之间可以用来转换,秒是基本单位。下面是常识：

- 1 = 1秒 （原文使用了两个==，可能有误 --译者注）
- 1分钟 = 60秒 （原文使用了两个==，可能有误 --译者注）
- 1小时 = 60分钟 （原文使用了两个==，可能有误 --译者注）
- 1天=24小时 （原文使用了两个==，可能有误 --译者注）
- 1周= 7天
- 1年= 365天

如果你使用这些单位执行日历计算，要注意以下问题。 因为闰秒，所以每年不总是等于365天,甚至每天也不是都有24小时,。由于无法预测闰秒,一个精确的日历库必须由外部oracle更新。

这些后缀不能用于变量。如果你想解释一些输入变量, 如天,你可以用以下方式:

```
function f(uint start, uint daysAfter) {

  if (now >= start + daysAfter * 1 days) { ... }}

Special Variables and Functions
```

特殊的变量和函数

有特殊的变量和函数总是存在于全局命名空间,主要用于提供关于blockchain的信息。

块和交易属性

- block.coinbase (address): :当前块的矿工的地址
- block.difficulty (uint):当前块的难度系数

- block.gaslimit (uint):当前块汽油限量
- block.number (uint):当前块编号
- block.blockhash (function(uint) returns (bytes32)):指定块的哈希值——最新的256个块的哈希值
- block.timestamp (uint):当前块的时间戳
- msg.data (bytes):完整的calldata
- msg.gas (uint):剩余的汽油
- msg.sender (address):消息的发送方(当前调用)
- msg.sig (bytes4):calldata的前四个字节(即函数标识符)
- msg.value (uint):所发送的消息中wei的数量
- now (uint):当前块时间戳(block.timestamp的别名)
- tx.gasprice (uint):交易的汽油价格
- tx.origin (address):交易发送方(完整的调用链)

请注意

msg的所有成员的值,包括msg.sender和msg.value可以在每个 external函数调用中改变。这包括调用库函数。

如果你想在库函数实现访问限制使用msg.sender, 你必须手动设置msg.sender作为参数。

请注意

由于所有块可伸缩性的原因，（所有）块的Hash值就拿不到。你只能访问最近的256块的Hash值,其他值为零。

数学和加密功能

```
addmod(uint x, uint y, uint k) returns (uint):
```

计算 (x + y) % k （按指定的精度，不能超过2**256）

mulmod(uint x, uint y, uint k) returns (uint):

compute (x *y) % k where the multiplication is performed with arbitrary precision and does not wrap around at 2256.* （按指定的精度，不能超过2*256）

计算compute (x * y) % k

```
sha3(...) returns (bytes32):
```

计算（紧凑排列的）参数的Ethereum-SHA-3 的Hash值值

```
sha256(...) returns (bytes32):
```

计算（紧凑排列的）参数的SHA-256 的Hash值

```
ripemd160(...) returns (bytes20):
```

计算（紧凑排列的）参数的 RIPEMD-160 的Hash值

```
ecrecover(bytes32, byte, bytes32, bytes32) returns (address):
```

恢复椭圆曲线特征的公钥-参数为(data, v, r, s)

在上述中，“紧凑排列”，意思是没有填充的参数的连续排列，也就是下面表达式是没有区别的

```
sha3("ab", "c")

sha3("abc")

sha3(0x616263)

sha3(6382179)

sha3(97, 98, 99)
```

如果需要填充，要用显示的形式表示： sha3(“x00x12”) 和 sha3(uint16(0x12))是相同的。

在一个私有的blockchain里，你可能（在使用）sha256, ripemd160 或 ecrecover (的时候) 碰到"Out-of-Gas"（的问题） 。原因在于这个仅仅是预编译的合约，合约要在他们接到的第一个消息以后才真正的生成（虽然他们的合约代码是硬编码的）。对于没有真正生成的合约的消息是非常昂贵的，这时就会碰到“Out-of-Gas”的问题。 这一问题的解决方法是事先把1wei 发送到各个你当前使用的各个合约上。这不是官方或测试网的问题。

合约相关的

```
this (current contract’s type):
```

当前的合约,显示可转换地址

```
selfdestruct(address)::
```

销毁当前合约,其资金发送给指定的地址

此外,当前合同的所有函数均可以被直接调用（包括当前函数）。

表达式和控制结构

控制结构

除了 switch和goto,solidity的绝大多数控制结构均来自于C / JavaScript，if, else, while, for, break, continue, return, ? :, 的语义均和C / JavaScript一样。

条件语句中的括号不能省略,但在单条语句前后的花括号可以省略。

注意,（Solidity中 ）没有象C和JavaScrip那样 ，从非布尔类型类型到布尔类型的转换, 所以if (1){...}不是合法的Solidity（语句）。

函数调用

内部函数调用

当前合约和函数可以直接调用(“内部”),也可以递归, 所以你会看到在这个“荒谬”的例子:

```
contract c {
  function g(uint a) returns (uint ret) { return f(); }
  function f() returns (uint ret) { return g(7) + f(); }}
```

这些函数调用在EMV里翻译成简单的jumps 语句。当前内存没有被清除, 即通过内存引用的函数是非常高效的。仅仅同一个合约的函数可在内部被调用。

外部函数调用

表达式this.g(8);也是一个合法的函数调用, 但是这一次,这个函数将被称为“外部”的, 通过消息调用,而不是直接通过jumps调用。其他合约的函数也是外部调用。对于外部调用,所有函数参数必须被复制到内存中。

当调用其他合约的函数时, 这个调用的wei的数量被发送和gas被定义:

```
contract InfoFeed {
  function info() returns (uint ret) { return 42; }
}
contract Consumer {
  InfoFeed feed;
  function setFeed(address addr) { feed = InfoFeed(addr); }
  function callFeed() { feed.info.value(10).gas(800)(); }
}
```

注意：表达式InfoFeed(addr)执行显式的类型转换, 意思是“我们知道给定的地址的合约类型是InfoFeed”, 这并不执行构造函数。 我们也可以直接使用函数setFeed(InfoFeed _feed) { feed = _feed; }。 注意： feed.info.value(10).gas(800)是(本地)设置值和函数调用发送的gas数量, 只有最后的括号结束后才完成实际的调用。

具名调用和匿名函数参数

有参数的函数调用可以有名字,不使用参数的名字(特别是返回参数)可以省略。

```
contract c {
  function f(uint key, uint value) { ... }
  function g() {
    // named arguments   具名参数
    f({value: 2, key: 3});
  }
  // omitted parameters 省略名字的参数
  function func(uint k, uint) returns(uint) {
    return k;
  }
}
```

表达式计算的次序

表达式的计算顺序是不确定的（准确地说是, 顺序表达式树中的子节点表达式计算顺序是不确定的的, 但他们对节点本身，计算表达式顺序当然是确定的)。只保证语句执行顺序，以及布尔表达式的短路规则。

赋值

析构赋值并返回多个值

Solidity内部允许元组类型,即一系列的不同类型的对象的大小在编译时是一个常量。这些元组可以用来同时返回多个值，并且同时将它们分配给多个变量(或左值运算)

```
contract C {
  uint[] data;
  function f() returns (uint, bool, uint) {
    return (7, true, 2);
  }
  function g() {
    // Declares and assigns the variables. Specifying the type explicitly is not possi
ble.
    var (x, b, y) = f();
    // Assigns to a pre-existing variable.
    (x, y) = (2, 7);
    // Common trick to swap values -- does not work for non-value storage types.
    (x, y) = (y, x);
    // Components can be left out (also for variable declarations).
    // If the tuple ends in an empty component,
    // the rest of the values are discarded.
    (data.length,) = f(); // Sets the length to 7
    // The same can be done on the left side.
    (,data[3]) = f(); // Sets data[3] to 2
    // Components can only be left out at the left-hand-side of assignments, with
    // one exception:
    (x,) = (1,);
    // (1,) is the only way to specify a 1-component tuple, because (1) is
    // equivalent to 1.
  }
}
contract C {
  uint[] data;
  function f() returns (uint, bool, uint) {
    return (7, true, 2);
  }
```

```
  function g() {
    // Declares and assigns the variables. Specifying the type explicitly is not possi
ble. 声明和赋值变量，不必显示定义类型
    var (x, b, y) = f();
    // Assigns to a pre-existing variable. 赋值给已经存在的变量
    (x, y) = (2, 7);
    // Common trick to swap values -- does not work for non-value storage types. 交换值
的技巧-对非值存储类型不起作用
    (x, y) = (y, x);
    // Components can be left out (also for variable declarations). 元素可排除（对变量声明
也适用）
    // If the tuple ends in an empty component, 如果元组是以空元素为结尾
    // the rest of the values are discarded.  值的其余部分被丢弃
    (data.length,) = f(); // Sets the length to 7 设定长度为7
    // The same can be done on the left side. 同样可以在左侧做
    (,data[3]) = f(); // Sets data[3] to 2  将data[3] 设为2
    // Components can only be left out at the left-hand-side of assignments, with
    // one exception:    组件只能在赋值的左边被排除，有一个例外
    (x,) = (1,);
    // (1,) is the only way to specify a 1-component tuple, because (1) is     (1,)是定
义一个元素的元组，（1）是等于1
    // equivalent to 1.
  }}
```

数组和结构体的组合

对于象数组和结构体这样的非值类型，赋值的语义更复杂些。赋值到一个状态变量总是需要创建一个独立的副本。另一方面,对基本类型来说，赋值到一个局部变量需要创建一个独立的副本, 即32字节的静态类型。如果结构体或数组(包括字节和字符串)从状态变量被赋值到一个局部变量, 局部变量则保存原始状态变量的引用。第二次赋值到局部变量不修改状态，只改变引用。赋值到局部变量的成员(或元素)将改变状态。

异常

有一些自动抛出异常的情况(见下文)。您可以使用throw 指令手动抛出一个异常。异常的影响是当前执行的调用被停止和恢复(即所有状态和余额的变化均没有发生)。另外， 异常也可以通过Solidity 函数 “冒出来”, (一旦“异常”发生, 就send "exceptions", call和callcode底层函数就返回false)。

捕获异常是不可能的。

在接下来的例子中,我们将演示如何轻松恢复一个Ether转移,以及如何检查send的返回值:

```
contract Sharer {
    function sendHalf(address addr) returns (uint balance) {
        if (!addr.send(msg.value/2))
            throw; // also reverts the transfer to Sharer
        return this.balance;
    }
}
contract Sharer {
    function sendHalf(address addr) returns (uint balance) {
        if (!addr.send(msg.value/2))
            throw; // also reverts the transfer to Sharer  也恢复Sharer的转移
        return this.balance;
    }
}
```

目前,Solidity异常自动发生,有三种情况,:

1. 如果你访问数组超出其长度 (即x[i] where i >= x.length)
2. 如果一个通过消息调用的函数没有正确的执行结束(即gas用完，或本身抛出异常)。
3. 如果一个库里不存在的函数被调用，或Ether被发送到一个函数库里。

在内部,当抛出异常时 ,Solidity就执行“非法jump”, 从而导致EMV(Ether虚拟机)恢复所有更改状态。这个原因是没有安全的方法可以继续执行, 预期的结果没有发生。由于我们想保留事务的原子性,（所以）最安全的做法是恢复所有更改，并使整个事务(或者至少调用)没有受影响。

合约

Solidity里的合约是面向对象语言里的类。它们持久存放在状态变量和函数中，（在里面）可以修改这些变量。在不同的合约（实例）中调用一个函数（的过程），（实际上）是在EVM（Ether虚拟机）中完成一次调用，并且完成（一次）上下文切换，（此时）状态变量是不可访问的。

创建合约

```
  合约可以从“外部”创建，也可以由Solidity合约创立。在创建合约时,它的构造函数(函具有与合约名称同名的函数)将被执行。
```

web3.js,即 JavaScript API， 是这样做的:

```
// The json abi array generated by the compiler

var abiArray = [

  {

    "inputs":[

      {"name":"x","type":"uint256"},

      {"name":"y","type":"uint256"}

    ],

    "type":"constructor"

  },

  {

    "constant":true,

    "inputs":[],

    "name":"x",

    "outputs":[{"name":"","type":"bytes32"}],

    "type":"function"

  }

];

var MyContract = web3.eth.contract(abiArray);// deploy new contractvar contractInstanc
e = MyContract.new(

  10,

  {from: myAccount, gas: 1000000}

);
```

```
// The json abi array generated by the compiler  由编译器生成的json abi 数组
var abiArray = [
  {
    "inputs":[
      {"name":"x","type":"uint256"},
      {"name":"y","type":"uint256"}
    ],
    "type":"constructor"
  },
  {
    "constant":true,
    "inputs":[],
    "name":"x",
    "outputs":[{"name":"","type":"bytes32"}],
    "type":"function"
  }
];
var MyContract = web3.eth.contract(abiArray);
// deploy new contract  部署一个新合约
var contractInstance = MyContract.new(
  10,
  {from: myAccount, gas: 1000000}
);
```

在内部，在合约的代码后要接着有构造函数的参数，但如果你使用web3.js，就不必关心这个。

如果是一个合约要创立另外一个合约，被创立的合约的源码（二进制代码）要能被创立者知晓。这意味着：循环创建依赖就成为不可能的事情。

```
contract OwnedToken {
    // TokenCreator is a contract type that is defined below.
    // It is fine to reference it as long as it is not used
    // to create a new contract.
    TokenCreator creator;
    address owner;
    bytes32 name;
    // This is the constructor which registers the
```

```
    // creator and the assigned name.
    function OwnedToken(bytes32 _name) {
        owner = msg.sender;
        // We do an explicit type conversion from `address`
        // to `TokenCreator` and assume that the type of
        // the calling contract is TokenCreator, there is
        // no real way to check that.
        creator = TokenCreator(msg.sender);
        name = _name;
    }
    function changeName(bytes32 newName) {
        // Only the creator can alter the name --
        // the comparison is possible since contracts
        // are implicitly convertible to addresses.
        if (msg.sender == creator) name = newName;
    }
    function transfer(address newOwner) {
        // Only the current owner can transfer the token.
        if (msg.sender != owner) return;
        // We also want to ask the creator if the transfer
        // is fine. Note that this calls a function of the
        // contract defined below. If the call fails (e.g.
        // due to out-of-gas), the execution here stops
        // immediately.
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }}
contract TokenCreator {
    function createToken(bytes32 name)
       returns (OwnedToken tokenAddress)
    {
        // Create a new Token contract and return its address.
        // From the JavaScript side, the return type is simply
        // "address", as this is the closest type available in
        // the ABI.
        return new OwnedToken(name);
```

```
    }
    function changeName(OwnedToken tokenAddress, bytes32 name) {
        // Again, the external type of "tokenAddress" is
        // simply "address".
        tokenAddress.changeName(name);
    }
    function isTokenTransferOK(
        address currentOwner,
        address newOwner
    ) returns (bool ok) {
        // Check some arbitrary condition.
        address tokenAddress = msg.sender;
        return (sha3(newOwner) & 0xff) == (bytes20(tokenAddress) & 0xff);
    }}
contract OwnedToken {
    // TokenCreator is a contract type that is defined below.  TokenCreator是在下面定义
的合约类型
    // It is fine to reference it as long as it is not used  若它本身不用于创建新的合约的话
，它就是一个引用
    // to create a new contract.
    TokenCreator creator;
    address owner;
    bytes32 name;
    // This is the constructor which registers the 这个是一个登记创立者和分配名称的结构函数
    // creator and the assigned name.
    function OwnedToken(bytes32 _name) {
        owner = msg.sender;
        // We do an explicit type conversion from `address` 我们做一次由`address`到`Toke
nCreator` 的显示类型转换，，确保调用合约的类型是 TokenCreator, （因为没有真正的方法来检测这一点）
        // to `TokenCreator` and assume that the type of
        // the calling contract is TokenCreator, there is
        // no real way to check that.
        creator = TokenCreator(msg.sender);
        name = _name;
    }
    function changeName(bytes32 newName) {
        // Only the creator can alter the name --  仅仅是创立者可以改变名称--
```

```
        // the comparison is possible since contracts  因为合约是隐式转换到地址上，这种比较是可能的
        // are implicitly convertible to addresses.
        if (msg.sender == creator) name = newName;
    }
    function transfer(address newOwner) {
        // Only the current owner can transfer the token.  仅仅是 仅仅是当前（合约）所有者可以转移 token
        if (msg.sender != owner) return;
        // We also want to ask the creator if the transfer  我们可以询问（合约）创立者"转移是否成功"
        // is fine. Note that this calls a function of the  注意下面定义的合约的函数调用
        // contract defined below. If the call fails (e.g.     如果函数调用失败，（如gas用完了等原因）
        // due to out-of-gas), the execution here stops  程序的执行将立刻停止
        // immediately.
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }}
contract TokenCreator {
    function createToken(bytes32 name)
       returns (OwnedToken tokenAddress)
    {
        // Create a new Token contract and return its address.  创立一个新的Token合约，并且返回它的地址
        // From the JavaScript side, the return type is simply  从 JavaScript观点看，返回的地址类型是"address"
        // "address", as this is the closest type available in   这个是和ABI最接近的类型
        // the ABI.
        return new OwnedToken(name);
    }
    function changeName(OwnedToken tokenAddress, bytes32 name) {
        // Again, the external type of "tokenAddress" is     "tokenAddress" 的外部类型也是 简单的"address".
        // simply "address".
        tokenAddress.changeName(name);
    }
    function isTokenTransferOK(
        address currentOwner,
        address newOwner
```

```
    ) returns (bool ok) {
        // Check some arbitrary condition. 检查各种条件
        address tokenAddress = msg.sender;
        return (sha3(newOwner) & 0xff) == (bytes20(tokenAddress) & 0xff);
    }
}
```

可见性和访问限制符

因为Solidity可以理解两种函数调用(“内部调用”，不创建一个真实的EVM调用(也称为“消息调用”)；“外部的调用”-要创建一个真实的EMV调用), 有四种的函数和状态变量的可见性。

函数可以被定义为external, public, internal or private，缺省是 public。对状态变量而言, external是不可能的,默认是 internal。

external: 外部函数是合约接口的一部分,这意味着它们可以从其他合约调用, 也可以通过事务调用。外部函数f不能被内部调用(即 f()不执行,但this.f()执行)。外部函数，当他们接收大数组时，更有效。

public:公共函数是合约接口的一部分,可以通过内部调用或通过消息调用。对公共状态变量而言，会有的自动访问限制符的函数生成(见下文)。

internal:这些函数和状态变量只能内部访问(即在当前合约或由它派生的合约),而不使用（关键字）this 。

private:私有函数和状态变量仅仅在定义该合约中可见， 在派生的合约中不可见。

请注意

在外部观察者中，合约的内部的各项均可见。用 *private* 仅仅防止其他合约来访问和修改（该合约中）信息, 但它对blockchain之外的整个世界仍然可见。

可见性说明符是放在在状态变量的类型之后，（也可以放在）参数列表和函数返回的参数列表之间。

```
contract c {
    function f(uint a) private returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

其他合约可以调用c.data()来检索状态存储中data的值,但不能访问（函数）f。由c派生的合约可以访问（合约中）setData（函数），以便改变data的值(仅仅在它们自己的范围里)。

访问限制符函数

编译器会自动创建所有公共状态变量的访问限制符功能。下文中的合约中有一个称作data的函数，它不带任何参数的，它返回一个uint类型, 状态变量的值是data。可以在声明里进行状态变量的初始化。

访问限制符函数有外部可见性。如果标识符是内部可访问(即没有this),则它是一个状态变量,如果外部可访问的(即 有this),则它是一个函数。

```
contract test {
    uint public data = 42;}
```

下面的例子复杂些：

```
contract complex {
    struct Data { uint a; bytes3 b; mapping(uint => uint) map; }
    mapping(uint => mapping(bool => Data[])) public data;}
```

它生成了如下形式的函数:

```
function data(uint arg1, bool arg2, uint arg3) returns (uint a, bytes3 b){
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;}
```

注意 结构体的映射省略了，因为没有好的方法来提供映射的键值。

函数修饰符

修饰符可以用来轻松改变函数的行为, 例如，在执行的函数之前自动检查条件。他们是可继承合约的属性，也可被派生的合约重写。

```
contract owned {
    function owned() { owner = msg.sender; }
    address owner;
    // This contract only defines a modifier but does not use
    // it - it will be used in derived contracts.
```

```
    // The function body is inserted where the special symbol

    // "_" in the definition of a modifier appears.

    // This means that if the owner calls this function, the

    // function is executed and otherwise, an exception is

    // thrown.

    modifier onlyowner { if (msg.sender != owner) throw; _ }}contract mortal is owned
{

    // This contract inherits the "onlyowner"-modifier from

    // "owned" and applies it to the "close"-function, which

    // causes that calls to "close" only have an effect if

    // they are made by the stored owner.

    function close() onlyowner {

        selfdestruct(owner);

    }}contract priced {

    // Modifiers can receive arguments:

    modifier costs(uint price) { if (msg.value >= price) _ }}contract Register is pric
ed, owned {

    mapping (address => bool) registeredAddresses;

    uint price;

    function Register(uint initialPrice) { price = initialPrice; }

    function register() costs(price) {

        registeredAddresses[msg.sender] = true;

    }

    function changePrice(uint _price) onlyowner {

        price = _price;

    }}

contract owned {

    function owned() { owner = msg.sender; }

    address owner;

    // This contract only defines a modifier but does not use 这个合约仅仅定义了修饰符，但
没有使用它

    // it - it will be used in derived contracts. 在派生的合约里使用

    // The function body is inserted where the special symbol  ,函数体插入到特殊的标识 "_"
定义的地方

    // "_" in the definition of a modifier appears.

    // This means that if the owner calls this function, the 这意味着若它自己调用此函数，则
函数将被执行

    // function is executed and otherwise, an exception is 否则，一个异常将抛出

    // thrown.
```

```
    modifier onlyowner { if (msg.sender != owner) throw; _ }
}
contract mortal is owned {
    // This contract inherits the "onlyowner"-modifier from   该合约是从"owned" 继承的"on
lyowner"修饰符，
    // "owned" and applies it to the "close"-function, which    并且应用到"close"函数，
如果他们存储owner
    // causes that calls to "close" only have an effect if
    // they are made by the stored owner.
    function close() onlyowner {
        selfdestruct(owner);
    }
}
contract priced {
    // Modifiers can receive arguments:  修饰符可以接收参数
    modifier costs(uint price) { if (msg.value >= price) _ }
}
contract Register is priced, owned {
    mapping (address => bool) registeredAddresses;
    uint price;
    function Register(uint initialPrice) { price = initialPrice; }
    function register() costs(price) {
        registeredAddresses[msg.sender] = true;
    }
    function changePrice(uint _price) onlyowner {
        price = _price;
    }
}
```

多个修饰符可以被应用到一个函数中（用空格隔开）,并顺序地进行计算。当离开整个函数时，显式返回一个修饰词或函数体, 同时在“_”之后紧接着的修饰符，直到函数尾部的控制流，或者是修饰体将继续执行。任意表达式允许修改参数，在修饰符中,所有函数的标识符是可见的。在此函数由修饰符引入的标识符是不可见的(虽然他们可以通过重写，改变他们的值)。

常量

状态变量可以声明为常量(在数组和结构体类型上仍然不可以这样做，映射类型也不可以)。

```
contract C {
    uint constant x = 32*\*22 + 8;
    string constant text = "abc";
}
```

编译器不保留这些变量存储块, 每到执行到这个语句时，常量值又被替换一次。

表达式的值只能包含整数算术运算。

回退函数

一个合约可以有一个匿名函数。若没有其他函数和给定的函数标识符一致的话，该函数将没有参数，将执行一个合约的调用(如果没有提供数据)。

此外,当合约接收一个普通的Ether时，函数将被执行(没有数据)。在这样一个情况下,几乎没有gas用于函数调用,所以调用回退函数是非常廉价的，这点非常重要。

```
contract Test {
    function() { x = 1; }
    uint x;}
//  This contract rejects any Ether sent to it. It is good
// practise to  include such a function for every contract
// in order not to loose  Ether.
contract Rejector {
    function() { throw; }
}
contract Caller {
  function callTest(address testAddress) {
      Test(testAddress).call(0xabcdef01); // hash does not exist
      // results in Test(testAddress).x becoming == 1.
      Rejector r = Rejector(0x123);
      r.send(2 ether);
      // results in r.balance == 0
  }
}
contract Test {
    function() { x = 1; }
    uint x;}
```

```
//  This contract rejects any Ether sent to it. It is good   这个合约拒绝任何发给它的Ether
.
// practise to  include such a function for every contract  为了严管Ether,在每个合约里包含一个这样的函数，是非常好的做法

// in order not to loose  Ether.

contract Rejector {

    function() { throw; }

}

contract Caller {

  function callTest(address testAddress) {

      Test(testAddress).call(0xabcdef01); // hash does not exist hash值不存在

      // results in Test(testAddress).x becoming == 1.  Test(testAddress).x的结果  becoming == 1

      Rejector r = Rejector(0x123);

      r.send(2 ether);

      // results in r.balance == 0     结果里r.balance == 0

  }

}
```

事件

事件允许EMV写日志功能的方便使用, 进而在dapp的用户接口中用JavaScript顺序调用,从而监听这些事件。

事件是合约中可继承的成员。当他们调用时，会在导致一些参数在事务日志上的存储--在blockchain上的一种特殊的数据结构。这些日志和合约的地址相关联, 将被纳入blockchain中，存储在block里以便访问(在**Frontier** 和 **Homestead**里是永久存储,但在**Serenity**里有些变化）。在合约内部，日志和事件数据是不可访问的(从创建该日志的合约里)。

SPV日志证明是可行的, 如果一个外部实体提供一个这样的证明给合约, 它可以检查blockchain内实际存在的日志(但要注意这样一个事实,最终要提供block的headers, 因为合约只能看到最近的256块hash值)。

最多有三个参数可以接收属性索引，它将对各自的参数进行检索： 可以对用户界面中的索引参数的特定值进行过滤。

如果数组(包括string和 bytes)被用作索引参数, 就会以sha3-hash形式存储，而不是topic。

除了用anonymous声明事件之外，事件的指纹的hash值都将是topic之一。这意味着，不可能通过名字来过滤特定的匿名事件。

所有非索引参数将被作为数据日志记录的一部分进行存储。

```
contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );
    function deposit(bytes32 _id) {
        // Any call to this function (even deeply nested) can
        // be detected from the JavaScript API by filtering
        // for `Deposit` to be called.
        Deposit(msg.sender, _id, msg.value);
    }
}
contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );
    function deposit(bytes32 _id) {
        // Any call to this function (even deeply nested) can 任何对这个函数的调用都能通过JavaScipt API ， 用`Deposit` 过滤来检索到（即使深入嵌套）
        // be detected from the JavaScript API by filtering
        // for `Deposit` to be called.
        Deposit(msg.sender, _id, msg.value);
    }
}
```

JavaScript API 的使用如下：

```
var abi = /\ abi as generated by the compiler /;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at(0x123 /\ address /);
var event = clientReceipt.Deposit();
// watch for changes
event.watch(function(error, result){
    // result will contain various information
    // including the argumets given to the Deposit
    // call.
    if (!error)
        console.log(result);});
// Or pass a callback to start watching immediately
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
var abi = /\ abi as generated by the compiler /;     /\ 由编译器生成的abi /;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at(0x123 /\ address /);   /\ 地址 /);
var event = clientReceipt.Deposit();
// watch for changes   观察变化
event.watch(function(error, result){
    // result will contain various information   结果包含不同的信息： 包括给Deposit调用的参数
    // including the argumets given to the Deposit
    // call.
    if (!error)
        console.log(result);});
// Or pass a callback to start watching immediately 或者通过callback立刻开始观察
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

底层日志的接口

还可以通过函数log0 log1,log2,log3 log4到 logi，共i+1个bytes32类型的参数来访问底层日志机制的接口。第一个参数将用于数据日志的一部分，其它的参数将用于topic。上面的事件调用可以以相同的方式执行。.

```
log3(
    msg.value,
    0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5adb20,
    msg.sender,
    _id
);
```

很长的十六进制数等于

sha3(“Deposit(address,hash256,uint256)”), 这个就是事件的指纹。

理解事件的额外的资源

- Javascipt文档
- 事件的用法举例
- 如何在js中访问

继承

通过包括多态性的复制代码，Solidity支持多重继承。

除非合约是显式给出的，所有的函数调用都是虚拟的，绝大多数派生函数可被调用。

即使合约是继承了多个其他合约, 在blockchain上只有一个合约被创建, 基本合约代码总是被复制到最终的合约上。

通用的继承机制非常类似于Python里的继承,特别是关于多重继承方面。

下面给出了详细的例子。

```
contract owned {
    function owned() { owner = msg.sender; }
    address owner;}
// Use "is" to derive from another contract. Derived// contracts can access all non-private members including// internal functions and state variables. These cannot be// accessed externally via `this`, though.contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
```

```
    }}

// These abstract contracts are only provided to make the// interface known to the com
piler. Note the function// without body. If a contract does not implement all// functi
ons it can only be used as an interface.contract Config {

    function lookup(uint id) returns (address adr);}contract NameReg {

    function register(bytes32 name);

    function unregister();

 }

// Multiple inheritance is possible. Note that "owned" is// also a base class of "mort
al", yet there is only a single// instance of "owned" (as for virtual inheritance in C
++).contract named is owned, mortal {

    function named(bytes32 name) {

        Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);

        NameReg(config.lookup(1)).register(name);

    }

    // Functions can be overridden, both local and

    // message-based function calls take these overrides

    // into account.

    function kill() {

        if (msg.sender == owner) {

            Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);

            NameReg(config.lookup(1)).unregister();

            // It is still possible to call a specific

            // overridden function.

            mortal.kill();

        }

    }}

// If a constructor takes an argument, it needs to be// provided in the header (or mod
ifier-invocation-style at// the constructor of the derived contract (see below)).contr
act PriceFeed is owned, mortal, named("GoldFeed") {

   function updateInfo(uint newInfo) {

      if (msg.sender == owner) info = newInfo;

   }

   function get() constant returns(uint r) { return info; }

   uint info;

}

contract owned {

    function owned() { owner = msg.sender; }

    address owner;}
```

```
// Use "is" to derive from another contract. Derived   用"is"是从其他的合约里派生出

// contracts can access all non-private members including   派生出的合约能够访问所有非私有的成员，包括内部函数和状态变量。  它们不能从外部用'this'来访问。

// internal functions and state variables. These cannot be

// accessed externally via `this`, though.

contract mortal is owned {

    function kill() {

        if (msg.sender == owner) selfdestruct(owner);

    }}

// These abstract contracts are only provided to make the  这些抽象的合约仅仅是让编译器知道已经生成了接口，

// interface known to the compiler. Note the function   注意：函数没有函数体。如果合约不做实现的话，它就只能当作接口。

// without body. If a contract does not implement all

// functions it can only be used as an interface.

contract Config {

    function lookup(uint id) returns (address adr);

}

contract NameReg {

    function register(bytes32 name);

    function unregister();

 }

// Multiple inheritance is possible. Note that "owned" is 多重继承也是可以的，注意"owned" 也是mortal的基类， 虽然 仅仅有"owned"的单个实例，（和C++里的virtual继承一样）

// also a base class of "mortal", yet there is only a single

// instance of "owned" (as for virtual inheritance in C++).

contract named is owned, mortal {

    function named(bytes32 name) {

        Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);

        NameReg(config.lookup(1)).register(name);

    }

    // Functions can be overridden, both local and  函数被重写，本地和基于消息的函数调用把这些override带入账户里。

    // message-based function calls take these overrides

    // into account.

    function kill() {

        if (msg.sender == owner) {

            Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);

            NameReg(config.lookup(1)).unregister();
```

```
            // It is still possible to call a specific  还可以调用特定的override函数
            // overridden function.
            mortal.kill();
        }
    }}
// If a constructor takes an argument, it needs to be  如果构造器里带有一个参数，有必要在头部给出，（或者在派生合约的构造器里使用修饰符调用方式modifier-invocation-style（见下文））
// provided in the header (or modifier-invocation-style at
// the constructor of the derived contract (see below)).
contract PriceFeed is owned, mortal, named("GoldFeed") {
   function updateInfo(uint newInfo) {
      if (msg.sender == owner) info = newInfo;
   }
   function get() constant returns(uint r) { return info; }
   uint info;
}
```

注意：在上文中，我们使用mortal.kill() 来“forward” 析构请求。这种做法是有问题的，请看下面的例子：

```
contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}
contract Base1 is mortal {
    function kill() { /\ do cleanup 1  清除1 / mortal.kill();
}
}
contract Base2 is mortal {
    function kill() { /\ do cleanup 2  清除2 / mortal.kill();
}
}
contract Final is Base1, Base2 {
}
```

Final.kill() 将调用Base2.kill作为最后的派生重写，但这个函数绕开了Base1.kill。因为它不知道有Base1。这种情况下要使用 super

```
contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}
contract Base1 is mortal {
    function kill() { /\ do cleanup 1   清除1  \/    super.kill(); }
}
contract Base2 is mortal {
    function kill() { /\ do cleanup 2  清除2  \/    super.kill(); }
}
contract Final is Base2, Base1 {
}
```

若Base1 调用了super函数，它不是简单地调用基本合约之一的函数， 它是调用最后继承关系的下一个基本合约的函数。所以它会调用 base2.kill（）（注意，最后的继承顺序是–从最后的派生合约开始：Final, Base1, Base2, mortal, owned）。当使用类的上下文中super不知道的情况下，真正的函数将被调用，虽然它的类型已经知道。这个和普通的virtual方法的查找相似。

基本构造函数的参数

派生的合约需要为基本构造函数提供所有的参数。这可以在两处进行：

```
contract Base {
    uint x;
    function Base(uint _x) { x = _x;}
}
contract Derived is Base(7) {
    function Derived(uint _y) Base(_y * _y) {
    }
}
```

第一种方式是直接在继承列表里实现（是 Base(7)），第二种方式是在派生的构造器的头部，修饰符被调用时实现（Base(_y * _y)）。如果构造函数参数是一个常量，并且定义了合约的行为或描述了它的行为，第一种方式比较方便。 如果基本构造函数参数依赖于派生合约的构造函数，则必须使用第二种方法。如果在这个荒谬的例子中，这两个地方都被使用，修饰符样式的参数优先。

多继承和线性化

允许多重继承的编程语言，要处理这样几个问题，其中一个是Diamond问题。Solidity是沿用Python的方式， 使用“C3线性化”，在基类的DAG强制使用特定的顺序。这导致单调但不允许有一些的继承关系。特别是，在其中的基础类的顺序是直接的，这点非常重要。在下面的代码中，Solidity会报错：“继承关系的线性化是不可能的”。

contract X {}

contract A is X {}

contract C is A, X {}

这个原因是，C要求X来重写A（定义A，X这个顺序），但A本身的要求重写X，这是一个矛盾，不能解决。

一个简单的规则是要指定基类中的顺序，从“最基本”到“最近派生”。

抽象契约

合约函数可以缺少实现（请注意，函数声明头将被终止），见下面的例子：

```
contract feline {

    function utterance() returns (bytes32);

}
```

这样的合约不能被编译（即使它们包含实现的函数和非实现的函数），但它们可以用作基本合约：

```
contract Cat is feline {

    function utterance() returns (bytes32) { return "miaow"; }

}
```

如果一个合约是从抽象合约中继承的，而不实现所有非执行功能，则它本身就是抽象的。

库

库和合约类似，但是它们的目的主要是在给定地址上部署，以及用EVM的CALLCODE特性来重用代码。这些代码是在调用合约的上下文里执行的，例如调用合约的指针和调用合约的存储能够被访问。由于库是一片独立的代码，如果它们显示地提供的话，就仅仅能访问到调用合约的状态变量（有方法命名它们）

下面的例子解释了怎样使用库（确保用using for 来实现）

```
library Set {

  // We define a new struct datatype that will be used to   我们定义了一个新的结构体数据类型，用于存放调用合约中的数据

  // hold its data in the calling contract.

  struct Data { mapping(uint => bool) flags; }

  // Note that the first parameter is of type "storage 注意第一个参数是 “存储引用”类型，这样仅仅是它的地址，而不是它的内容在调用中被传入 这是库函数的特点，

  // reference" and thus only its storage address and not

  // its contents is passed as part of the call.  This is a

  // special feature of library functions.  It is idiomatic  若第一个参数用"self"调用时很笨的的，如果这个函数可以被对象的方法可见。

  // to call the first parameter 'self', if the function can

  // be seen as a method of that object.

  function insert(Data storage self, uint value)

      returns (bool)

  {

      if (self.flags[value])

          return false; // already there 已经在那里

      self.flags[value] = true;

      return true;

  }

  function remove(Data storage self, uint value)

      returns (bool)

  {

      if (!self.flags[value])

          return false; // not there 不在那里

      self.flags[value] = false;

      return true;

  }

  function contains(Data storage self, uint value)
```

```
        returns (bool)
    {
        return self.flags[value];
    }
}
contract C {
    Set.Data knownValues;
    function register(uint value) {
        // The library functions can be called without a  这个库函数没有特定的函数实例被调用，因为“instance”是当前的合约
        // specific instance of the library, since the
        // "instance" will be the current contract.
        if (!Set.insert(knownValues, value))
            throw;
    }
    // In this contract, we can also directly access knownValues.flags, if we want 在这个合约里，如果我们要的话，也可以直接访问 knownValues.flags
.*}
```

当然，你不必这样使用库--他们也可以事前不定义结构体数据类型，就可以使用。 没有任何存储引入参数，函数也可以执行。也可以在任何位置，有多个存储引用参数。

Set.contains, Set.insert and Set.remove都可编译到（CALLCODE）外部合约/库。如果你使用库，注意真正进行的外部函数调用，所以`msg.sender不再指向来源的sender了，而是指向了正在调用的合约。msg.value包含了调用库函数中发送的资金。

因为编译器不知道库将部署在哪里。这些地址不得不由linker填进最后的字节码（见使用命令行编译器如何使用命令行编译器链接）。如果不给编译器一个地址做参数，编译的十六进制码就会包含**Set** 这样的占位符（Set是库的名字）。通过替换所有的40个字符的十六进制编码的库合约的地址，地址可以手动进行填充。

比较合约和库的限制：

- 无状态变量
- 不能继承或被继承

（这些可能在以后会被解除）

库的常见“坑”

msg.sender的值

msg.sender的值将是调用库函数的合约的值。

例如，如果A调用合约B，B内部调用库C。在库C库的函数调用里，msg.sender将是合约B的地址。

表达式LibraryName.functionName() 用CALLCODE完成外部函数调用， 它映射到一个真正的EVM调用，就像otherContract.functionName() 或者 this.functionName()。这种调用可以一级一级扩展调用深度（最多1024级），把msg.sender存储为当前的调用者，然后执行库合约的代码，而不是执行当前的合约存储。这种执行方式是发生在一个完全崭新的内存环境中，它的内存类型将被复制，并且不能绕过引用。

转移Ether

原则上使用LibraryName.functionName.value(x)(）来转移Ether。但若使用CALLCODE，Ether会在当前合约里用完。

Using For

指令 using A for B; 可用于附加库函数（从库A）到任何类型（B）。这些函数将收到一个作为第一个参数的对象（像Python中self变量）。

using A for *;，是指函数从库A附加到任何类型。

在这两种情况下，所有的函数将被附加，（即使那些第一个参数的类型与对象的类型不匹配）。该被调用函数的入口类型将被检查，并进行函数重载解析。

using A for B; 指令在当前的范围里是有效的，作用范围限定在现在的合约里。但（出了当前范围）在全局范围里就被移除。因此，通过 including一个模块，其数据类型（包括库函数）都将是可用的，而不必添加额外的代码。

让我们用这种方式重写库中的set示例：

```
// This is the same code as before, just without comments

library Set {

  struct Data { mapping(uint => bool) flags; }

  function insert(Data storage self, uint value)

      returns (bool)

  {

      if (self.flags[value])

        return false; // already there
```

```
        self.flags[value] = true;
        return true;
    }
    function remove(Data storage self, uint value)
        returns (bool)
    {
        if (!self.flags[value])
            return false; // not there
        self.flags[value] = false;
        return true;
    }
    function contains(Data storage self, uint value)
        returns (bool)
    {
        return self.flags[value];
    }
}
contract C {
    using Set for Set.Data; // this is the crucial change
    Set.Data knownValues;
    function register(uint value) {
        // Here, all variables of type Set.Data have
        // corresponding member functions.
        // The following function call is identical to
        // Set.insert(knownValues, value)
        if (!knownValues.insert(value))
            throw;
    }
}
// This is the same code as before, just without comments   这个代码和之前的一样，仅仅是没有注释
library Set {
  struct Data { mapping(uint => bool) flags; }
  function insert(Data storage self, uint value)
      returns (bool)
  {
      if (self.flags[value])
```

```
        return false; // already there 已经在那里
      self.flags[value] = true;
      return true;
  }
  function remove(Data storage self, uint value)
      returns (bool)
  {
      if (!self.flags[value])
          return false; // not there 没有
      self.flags[value] = false;
      return true;
  }
  function contains(Data storage self, uint value)
      returns (bool)
  {
      return self.flags[value];
  }
}
contract C {
    using Set for Set.Data; // this is the crucial change   这个是关键的变化
    Set.Data knownValues;
    function register(uint value) {
        // Here, all variables of type Set.Data have   这里，所有Set.Data 的变量都有相应的成员函数
        // corresponding member functions.
        // The following function call is identical to  下面的函数调用和Set.insert(knownValues, value) 作用一样
        // Set.insert(knownValues, value)
        if (!knownValues.insert(value))
            throw;
    }
}
It is also possible to extend elementary types in that way:
这个也是一种扩展基本类型的（方式）
library Search {
    function indexOf(uint[] storage self, uint value) {
        for (uint i = 0; i < self.length; i++)
```

```
            if (self[i] == value) return i;
        return uint(-1);
    }}
contract C {
    using Search for uint[];
    uint[] data;
    function append(uint value) {
        data.push(value);
    }
    function replace(uint _old, uint _new) {
        // This performs the library function call   这样完成了库函数的调用
        uint index = data.find(_old);
        if (index == -1)
            data.push(_new);
        else
            data[index] = _new;
    }}
```

注意：所有的库函数调用都是调用实际的EVM。这意味着，如果你要使用内存或值类型，就必须执行一次拷贝操作，即使是self变量。拷贝没有完成的情况可能是存储引用变量已被使用。

Next Previous

杂项

存储中状态变量的布局

静态尺寸大小的变量（除了映射和动态尺寸大小的数组类型（的其他类型变量））在存储中，是从位置0连续存储。如果可能的话，不足32个字节的多个条目被紧凑排列在一个单一的存储块，参见以下规则：

- 在存储块中的第一项是存储低阶对齐的。
- 基本类型只使用了正好存储它们的字节数。
- 如果一个基本类型不适合存储块的剩余部分，则移动到下一个存储块中。
- 结构和数组的数据总是开始一个新的块并且占整个块（根据这些规则，结构或数组项都是紧凑排列的）。

结构和数组元素是一个接着一个存储排列的，就如当初它们被声明的次序。

由于无法预知的分配的大小，映射和动态尺寸大小的数组类型（这两种类型）是使用sha3 计算来找到新的起始位置，来存放值或者数组数据。这些起始位置总是满栈块。

根据上述规则，映射或动态数组本身存放在（没有填满）的存储块位置p（或从映射到映射或数组递归应用此规则）。对于一个动态数组，存储块存储了数组元素的数目（字节数组和字符串是一个例外，见下文）。对于映射，存储块是未使用的（但它是需要的，因此，前后相邻的两个相同的映射，将使用一个不同的hash分布）。数组数据位于sha3(p)， 对应于一个映射key值k位于 sha3(k . p) （这里 . 是连接符）。如果该值又是一个非基本类型，位置的偏移量是sha3(k . p)。

如果bytes 和 string是短类型的，它们将和其长度存储在同一个存储块里。特别是：如果数据最长31字节，它被存储在高阶字节（左对齐）, 低字节存储length *2*。 如果是长类型，主存储块存储 *length* 2 + 1, 数据存储在sha3(shot)。

因此，本合约片段如下：

```
contract c {
  struct S { uint a; uint b; }
  uint x;
  mapping(uint => mapping(uint => S)) data;
}
```

深奥的特点

在Solidity的类型系统中，有一些在语法中没有对应的类型。其中就有函数的类型。但若使用var（这个关键字），该函数就被认为是这个类型的局部变量：

```
contract FunctionSelector {
  function select(bool useB, uint x) returns (uint z) {
    var f = a;
    if (useB) f = b;
    return f(x);
  }
  function a(uint x) returns (uint z) {
    return x * x;
  }
  function b(uint x) returns (uint z) {
    return 2 * x;
  }
}
```

（在上面的程序片段中）

若调用select(false, x)，就会计算 x *x* 。若调用*select(true, x)*）就会计算 *2* x。

内部-优化器

Solidity 优化器是在汇编级别上的操作，所以它也可以同时被其他语言所使用。它将指令的（执行）次序，在JUMP 和 JUMPDEST上分成基本的块。在这些块中，指令被解析 。 堆栈、内存或存储上的每一次修改，都将作为表达式被记录。该表达式包括一条指令以及指向其他表达式的一系列参数的一个指针。现在的主要意思是要找到相等的表达式（在每次输入），做成了表达式的类。优化器首先在已知的表达式列表里找，若找不到的话，就根据constant + constant = sum_of_constants 或 X * 1 = X 来简化。 因为这样做是递归的，如果第二个因子是一个更复杂的表达式，我们也可以应用latter规则来计算。存储和内存位置的修改，是不知道存储和内存的位置的区别。如果我们先写到的位置x，再写到位置y , x,y均是输入变量。第二个可以覆写第一个，所以我们不知道x是存放在y之后的。另一方面，如果一个简化的表达式 x-y 能计算出一个非零常数，我们就知道x存放的内容。

在这个过程结束时，我们知道，表达式必须在堆栈中结尾，并有一系列对内存和存储的修改。这些信息存储在block上，并链接这些block。此外，有关堆栈，存储和内存配置的信息会转发到下一个block。如果我们知道所有的 JUMP 和 JUMPI 指令，我们可以建立一个完整的

程序控制流图。如果我们不知道目标块（原则上，跳转目标是从输入里得到的），我们必须清除所有输入状态的存储块上的信息，（因为它的目标块未知）。如果条件计算的结果为一个常量，它转化为一个无条件jump。

作为最后一步，在每个块中的代码完全可以重新生成。从堆栈里block的结尾表达式开始，创建一个依赖关系图。每个不是这个图上的操作将舍弃。现在能按照原来代码的顺序，生成对存储和内存修改的代码（舍弃不必要的修改）。最后，在正确位置的堆栈上，生成所有的值。

这些步骤适用于每一个基本块， 如果它是较小的，用新生成的代码来替换。如果一个基本块在JUMPI上进行分割，在分析过程中，条件表达式的结果计算为一个常数，JUMP就用常量值进行替换。代码如下

```
var x = 7;

data[7] = 9;

if (data[x] != x + 2)

  return 2;

else

  return 1;
```

简化成下面可以编译的形式

```
data[7] = 9;

return 1;
```

即使在开始处包含有jump指令

使用命令行编译器

一个 Solidity 库构建的目标是solc， Solidity命令行编译器。使用solc –help 为您提供所有选项的解释。编译器可以产生不同的输出，从简单的二进制文件，程序集的抽象语法树（解析树）到gas使用的估量。如果你只想编译一个文件，你运行solc –bin sourceFile.sol ， 将会打印出二进制。你部署你的合约之前，使用solc –optimize –bin sourceFile.sol 来激活优化器。如果你想获得一些solc更进一步的输出变量，可以使用solc -o outputDirectory –bin –ast –asm sourceFile.sol，（这条命令）将通知编译器输出结果到单独的文件中。

命令行编译器会自动从文件系统中读取输入文件，但也可以如下列方法，提供重定向路径prefix=path ：

solc github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/ =/usr/local/lib/fallback file.sol

该命令告诉编译器在/usr/local/lib/dapp-bin目录下，寻找以github.com/ethereum/dapp-bin/ 开头的文件，如果找不到的话，到usr/local/lib/fallback目录下找（空前缀总是匹配）。

solc不会从remapping目标的外部，或者显式定义的源文件的外部文件系统读取文件，所以要写成 import “/etc/passwd”; 只有增加 =/ 作为remapping,程序才能工作。

如果remapping里找到了多个匹配，则选择有共同的前缀最长的那个匹配。

如果你的合约使用了 libraries ，你会注意到字节码中包含了 form LibraryName 这样的子字符串。你可以在这些地方使用solc 作为链接器，来插入库地址 ：

Either add –libraries “Math:0x12345678901234567890 Heap:0xabcdef0123456” 提供每个库的地址， 或者在文件中存放字符串（每行一个库）

然后运行solc，后面写 –libraries fileName.

如果solc后面接着 –link 选项，所有输入文件将被解释为未链接的二进制文件（十六进制编码）， LibraryName形式如前所述， 库此时被链接（从stdin读取输入，从stdout输出）。在这种情况下，除了–libraries,其他所有的选项都将被忽略（包括 -o)

提示和技巧

- 在数组中使用delete，就是删除数组中的所有元素。
- 使用较短的类型和结构元素，短类型分组在一起进行排序。sstore操作可能合并成一个单一的sstore，这可以降低gas的成本（sstore消耗5000或20000 gas，所以这是你必须优化的原因）。使用天gas的价格估算功能（优化器 enable）进行检查！
- 让你的状态变量公开，编译器会免费创建 getters 。
- 如果你结束了输入或状态的检查条件，请尝试使用函数修饰符。
- 如果你的合约有一个功能send， 但你想使用内置的send功能，请使用address(contractVariable).send(amount)。
- 如果你不想你的合约通过send接收ether，您可以添加一个抛出回退函数 function() { throw; }.。
- 用单条赋值语句初始化存储结构：x = MyStruct({a: 1, b: 2});

陷阱

不幸的是，还有一些编译器微妙的情况还没有告诉你。

- 在for (var i = 0; i < arrayName.length; i++) { ... }, i的类型是uint8，因为这是存放值0最小的类型。如果数组元素超过255个，则循环将不会终止。

列表

全局变量

- block.coinbase (address):当前块的矿场的地址
- block.difficulty (uint):当前块的难度
- block.gaslimit (uint):当前块的gaslimit
- block.number (uint):当前块的数量
- block.blockhash (function(uint) returns (bytes32)):给定的块的hash值, 只有最近工作的256个块的hash值
- block.timestamp (uint):当前块的时间戳
- msg.data (bytes):完整的calldata
- msg.gas (uint): 剩余gas
- msg.sender (address):消息的发送者（当前调用）
- msg.value (uint):和消息一起发送的wei的数量
- now (uint):当前块的时间戳（block.timestamp的别名）
- tx.gasprice (uint):交易的gas价格
- tx.origin (address):交易的发送者（全调用链）
- sha3(...) returns (bytes32):计算（紧凑排列的）参数的 Ethereum-SHA3 hash值
- sha256(...) returns (bytes32)计算（紧凑排列的）参数的SHA256 hash值
- ripemd160(...) returns (bytes20):计算 256个（紧凑排列的）参数的RIPEMD
- ecrecover(bytes32, uint8, bytes32, bytes32) returns (address):椭圆曲线签名公钥恢复
- addmod(uint x, uint y, uint k) returns (uint):计算（x + y）K，加法为任意精度，不以2 ** 256取余
- mulmod(uint x, uint y, uint k) returns (uint):计算（XY）K，乘法为任意精度，不以2 * 256取余
- this (current contract's type): 当前合约，在地址上显式转换
- super:在层次关系上一层的合约
- selfdestruct(address):销毁当前的合同，将其资金发送到指定address地址

- *.balance:address*地址中的账户余额（以*wei*为单位）
- *.send(uint256) returns (bool):*将一定量*wei*发送给*address*地址，若失败返回*false*。

函数可见性定义符

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

- public:在外部和内部均可见（创建存储/状态变量的访问者函数）
- private:仅在当前合约中可见
- external: 只有外部可见（仅对函数）- 仅仅在消息调用中（通过this.fun）
- internal: 只有内部可见

Modifiers

- constant for state variables: Disallows assignment (except initialisation), does not occupy storage slot.
- constant for functions: Disallows modification of state - this is not enforced yet.
- anonymous for events: Does not store event signature as topic.
- indexed for event parameters: Stores the parameter as topic.

修饰符

- constant for state variables: 不允许赋值（除了初始化），不占用存储块。
- constant for functions:不允许改变状态- 这个目前不是强制的。
- anonymous for events:不能将topic作为事件指纹进行存储。
- indexed for event parameters: 将topic作为参数存储。

原文：http://solidity.readthedocs.io/en/latest/security-considerations.html

编程规范

概述

本指南用于提供编写Solidity的编码规范，本指南会随着后续需求不断修改演进，可能会增加新的更合适的规范，旧的不适合的规范会被废弃。

当然，很多项目可能有自己的编码规范，如果存在冲突，请参考项目的编码规范。

本指南的结构及规范建议大都来自于python的pep8编码规范。

本指南不是说必须完全按照指南的要求进行solidity编码，而是提供一个总体的一致性要求，这个和pep8的理念相似（译注：pep8的理念大概是：强制的一致性是非常愚蠢的行为，参见：pep8）。

本指南是为了提供编码风格的一致性，因此一致性这一理念是很重要的，在项目中编码风格的一致性更加重要，而在同一个函数或模块中风格的一致性是最重要的。而最最重要的是：你要知道什么时候需要保持一致性，什么时候不需要保持一致性，因为有时候本指南不一定适用，你需要根据自己的需要进行权衡。可以参考下边的例子决定哪一种对你来说是最合适的。

代码布局

缩进

每行使用4个空格缩进

tab或空格

空格是首选缩进方式

禁止tab和空格混合使用

回车（空行）

两个合约之间增加两行空行

规范的方式：

```
contract A {
    ...}
contract B {
    ...}
contract C {
    ...}
```

不规范的方式：

```
contract A {
    ...}contract B {
    ...}
contract C {
    ...}
```

合约内部函数之间需要回车，如果是函数声明和函数实现一起则需要两个回车

规范的方式：

```
contract A {
    function spam();
    function ham();
}
contract B is A {
    function spam() {
        ...
    }
    function ham() {
        ...
    }
}
```

不规范的方式：

```
contract A {
    function spam() {
        ...
    }
    function ham() {
        ...
    }}
```

源文件编码方式

首选UTF-8或者ASCII编码

引入

一般在代码开始进行引入声明

规范的方式：

```
import "owned";
contract A {
    ...}
contract B is owned {
    ...}
```

不规范的方式：

```
contract A {
    ...}
import "owned";
contract B is owned {
    ...}
```

表达式中的空格使用方法

以下场景避免使用空格

- 括号、中括号，花括号之后避免使用空格

Yes规范的方式: spam(ham[1], Coin({name: “ham”}));

No不规范的方式: spam(ham[1], Coin({ name: “ham” }));

- 逗号和分号之前避免使用空格

Yes规范的方式: function spam(uint i, Coin coin);

No不规范的方式: function spam(uint i , Coin coin) ;

- 赋值符前后避免多个空格

规范的方式：

```
x = 1;
y = 2;
long_variable = 3;
```

不规范的方式：

```
x             = 1;
y             = 2;
long_variable = 3;
```

控制结构

合约、库。函数、结构体的花括号使用方法：

- 左花括号和声明同一行
- 右括号和左括号声明保持相同缩进位置。
- 左括号后应回车

规范的方式：

```
contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

不规范的方式：

```
contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

以上建议也同样适用于if、else、while、for。

此外，if、while、for条件语句之间必须空行

规范的方式：

```
if (...) {
    ...
}
for (...) {
    ...
}
```

不规范的方式：

```
if (...)
{
    ...
}
while(...)
{
}
for (...)
 {
    ...;
}
```

对于控制结构内部如果只有单条语句可以不需要使用括号。

规范的方式：

```
if (x < 10)
    x += 1;
```

不规范的方式：

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

对于if语句如果包含else或者else if语句，则else语句要新起一行。else和else if的内部规范和if相同。

规范的方式：

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
if (x < 3)
    x += 1;
else
    x -= 1;
```

不规范的方式：

```
if (x < 3) {
    x += 1;}
else {
    x -= 1;}
```

函数声明

对于简短函数声明，建议将函数体的左括号和函数名放在同一行。

右括号和函数声明保持相同的缩进。

左括号和函数名之间要增加一个空格。

规范的方式：

```
function increment(uint x) returns (uint) {

    return x + 1;

}

function increment(uint x) public onlyowner returns (uint) {

    return x + 1;

}
```

不规范的方式：

```
function increment(uint x) returns (uint)

{

    return x + 1;

}

function increment(uint x) returns (uint)

{

    return x + 1;

}

function increment(uint x) returns (uint)

 {

    return x + 1;

}

function increment(uint x) returns (uint)

{

    return x + 1;

}
```

默认修饰符应该放在其他自定义修饰符之前。

规范的方式：

```
function kill() public onlyowner {
    selfdestruct(owner);
}
```

不规范的方式：

```
function kill() onlyowner public {
    selfdestruct(owner);
}
```

对于参数较多的函数声明可将所有参数逐行显示，并保持相同的缩进。函数声明的右括号和函数体左括号放在同一行，并和函数声明保持相同的缩进。

规范的方式：

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f,
) {
    do_something;
}
```

不规范的方式：

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) {
    do_something;
}
function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) {
    do_something;
}
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) {
    do_something;
}
```

如果函数包括多个修饰符，则需要将修饰符分行并逐行缩进显示。函数体左括号也要分行。

规范的方式：

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address)
{
    do_something;
}
function thisFunctionNameIsReallyLong(
    address x,
    address y,
    address z,)
    public
    onlyowner
    priced
    returns (address)
{
    do_something;
}
```

不规范的方式：

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
                                      public
                                      onlyowner
                                      priced
                                      returns (address) {
    do_something;
}
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public onlyowner priced returns (address){
    do_something;
}
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address) {
    do_something;
}
```

对于需要参数作为构造函数的派生合约，如果函数声明太长或者难于阅读，建议将其构造函数中涉及基类的构造函数分行独立显示。

规范的方式：

```
contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
    {
        // do something with param5
    }
}
```

不规范的方式：

```
contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4)
    {
        // do something with param5
    }
}
contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4) {
        // do something with param5
    }
}
```

对于函数声明的编程规范主要用于提升可读性，本指南不可能囊括所有编程规范，对于不涉及的地方，程序猿可发挥自己的主观能动性。

映射 待完成

变量声明

对于数组变量声明，类型和数组中括号直接不能有空格。

规范的方式: uint[] x; 不规范的方式: uint [] x;

其他建议

- 赋值运算符两边要有一个空格

规范的方式：

```
x = 3;x = 100 / 10;x += 3 + 4;x |= y && z;
```

不规范的方式：

```
x=3;x = 100/10;x += 3+4;x |= y&&z;
```

- 为了显示优先级，优先级运算符和低优先级运算符之间要有空格，这也是为了提升复杂声明的可读性。对于运算符两侧的空格数目必须保持一致。

规范的方式：

```
x = 2**3 + 5;x = 2***y + 3*z;x = (a+b) * (a-**b);
```

不规范的方式：

```
x = 2** 3 + 5;x = y+z;x +=1;
```

命名规范

命名规范是强大且广泛使用的，使用不同的命名规范可以传递不同的信息。

以下建议是用来提升代码的可读性，因此被规范不是规则而是用于帮助更好的解释相关代码。

最后，编码风格的一致性是最重要的。

命名方式

为了防止混淆，以下命名用于说明（描述）不同的命名方式。

- b（单个小写字母）
- B（单个大写字母）
- 小写
- 有下划线的小写
- 大写
- 有下划线的大写
- CapWords规范（首字母大写）
- 混合方式（与CapitalizedWords的不同在于首字母小写!）
- 有下划线的首字母大写（译注：对于python来说不建议这种方式）

注意

当使用CapWords规范（首字母大写）的缩略语时，缩略语全部大写，比如HTTPServerError比HttpServerError就好理解一点。

避免的命名方式

- l - Lowercase letter el 小写的l

- O - Uppercase letter oh 大写的o
- I - Uppercase letter eye 大写的i

永远不要用字符‘l'(小写字母el(就是读音，下同))，‘O'(大写字母oh)，或‘I'(大写字母eye)作为单字符的变量名。在某些字体中这些字符不能与数字1和0分辨。试着在使用‘l'时用‘L'代替。

合约及库的命名

合约应该使用CapWords规范命名（首字母大写）。

事件

事件应该使用CapWords规范命名（首字母大写）。

函数命名

函数名使用大小写混合

函数参数命名

当定义一个在自定义结构体上的库函数时，结构体的名称必须具有自解释能力。

局部变量命名

大小写混合

常量命名

常量全部使用大写字母并用下划线分隔。

修饰符命名

功能修饰符使用小写字符并用下划线分隔。

避免冲突

- 单个下划线结尾

当和内置或者保留名称冲突时建议使用本规范。

通用建议

待完成

通用模式

访问限制

访问限制是合约的一种通用模式，但你不能限制任何人获取你的合约和交易的状态。当然，你可以通过加密来增加读取难度，但是如果你的合约需要读取该数据（指加密的数据），其他人也可以读取。

你可以通过将合约状态设置为私有来限制其他合约来读取你的合约状态。

此外，你可以限制其他人修改你的合约状态或者调用你的合约函数，这也是本章将要讨论的。

函数修饰符的使用可以让这些限制（访问限制）具有较好的可读性。

```
contract AccessRestriction {
    // These will be assigned at the construction
    // phase, where `msg.sender` is the account
    // creating this contract.
    //以下变量将在构造函数中赋值
    //msg.sender是你的账户
    //创建本合约
    address public owner = msg.sender;
    uint public creationTime = now;

    // Modifiers can be used to change
    // the body of a function.
    // If this modifier is used, it will
    // prepend a check that only passes
    // if the function is called from
    // a certain address.
```

//修饰符可以用来修饰函数体，如果使用该修饰符，当该函数被其他地址调用时将会先检查是否允许调用（译注：就是说外部要调用本合约有修饰符的函数时会检查是否允许调用，比如该函数是私有的则外部不能调用。）

```
    modifier onlyBy(address _account)
    {
        if (msg.sender != _account)
            throw;
        // Do not forget the "_"! It will
        // be replaced by the actual function
        // body when the modifier is invoked.
        //account变量不要忘了“_”
        _
    }

    /// Make `_newOwner` the new owner of this
    /// contract.
   //修改当前合约的宿主
    function changeOwner(address _newOwner)
        onlyBy(owner)
    {
        owner = _newOwner;
    }

    modifier onlyAfter(uint _time) {
        if (now < _time) throw;
        _
    }

    /// Erase ownership information.
    /// May only be called 6 weeks after
    /// the contract has been created.
   //清除宿主信息。只能在合约创建6周后调用
    function disown()
        onlyBy(owner)
        onlyAfter(creationTime + 6 weeks)
    {
        delete owner;
    }

    // This modifier requires a certain
    // fee being associated with a function call.
    // If the caller sent too much, he or she is
    // refunded, but only after the function body.
    // This is dangerous, because if the function
    // uses `return` explicitly, this will not be
    // done!
    //该修饰符和函数调用关联时需要消耗一部分费用。调用者发送的多余费用会在函数执行完成后返还，但这个是
相当危险的，因为如果函数
    modifier costs(uint _amount) {
        if (msg.value < _amount)
            throw;
        _
        if (msg.value > _amount)
            msg.sender.send(_amount - msg.value);
    }

    function forceOwnerChange(address _newOwner)
        costs(200 ether)
    {
        owner = _newOwner;
        // just some example condition
        if (uint(owner) & 0 == 1)
            // in this case, overpaid fees will not
            // be refunded
            return;
        // otherwise, refund overpaid fees
    }}
```

A more specialised way in which access to function calls can be restricted will be discussed in the next example.

State Machine Contracts often act as a state machine, which means that they have certain stages in which they behave differently or in which different functions can be called. A function call often ends a stage and transitions the contract into the next stage (especially if the contract models interaction). It is also common that some stages are automatically reached at a certain point in time.

An example for this is a blind auction contract which starts in the stage “accepting blinded bids”, then transitions to “revealing bids” which is ended by “determine auction autcome”.

Function modifiers can be used in this situation to model the states and guard against incorrect usage of the contract.

Example

In the following example, the modifier atStage ensures that the function can only be called at a certain stage.

Automatic timed transitions are handled by the modifier timeTransitions, which should be used for all functions.

Note

Modifier Order Matters. If atStage is combined with timedTransitions, make sure that you mention it after the latter, so that the new stage is taken into account.

Finally, the modifier transitionNext can be used to automatically go to the next stage when the function finishes.

Note

Modifier May be Skipped. Since modifiers are applied by simply replacing code and not by using a function call, the code in the transitionNext modifier can be skipped if the function itself uses return. If you want to do that, make sure to call nextStage manually from those functions.

```
contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
        AreWeDoneYet,
        Finished
    }
    // This is the current stage.
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = now;

    modifier atStage(Stages _stage) {
        if (stage != _stage) throw;
        _
    }
    function nextStage() internal {
```

```
        stage = Stages(uint(stage) + 1);
    }
    // Perform timed transitions. Be sure to mention
    // this modifier first, otherwise the guards
    // will not take the new stage into account.
    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindedBids &&
                    now >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
                now >= creationTime + 12 days)
            nextStage();
        // The other stages transition by transaction
    }

    // Order of the modifiers matters here!
    function bid()
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // We will not implement that here
    }
    function reveal()
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // This modifier goes to the next stage
    // after the function is done.
    // If you use `return` in the function,
    // `nextStage` will not be called
    // automatically.
    modifier transitionNext()
    {
        _
        nextStage();
    }
    function g()
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
        // If you want to use `return` here,
        // you have to call `nextStage()` manually.
    }
    function h()
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }
    function i()
        timedTransitions
        atStage(Stages.Finished)
    {
    }}
```

Next Previous

常见问题

This list was originally compiled by fivedogit)gmail.com)).

基础问题

Solidity是什么？

Solidity是受Javascript启发的编程语言，可以被用来在以太坊区块链上创建智能合约。还有其它编程语言（LLL，Serpent等）也可以创建智能合约。Solidity更被开发者喜爱的主要原因是它是静态类型语言，提供许多高级特性，例如继承、函数库、用户定义的复杂类型和字节码优化。

Solidity合约能够以不同方式被编译（见下文），输出结果可以被剪贴/复制到一个geth控制台，然后被部署到以太坊区块链。

Fivedogit写了一些合约例子，Solidity的每个特性都应该有对应的测试合约。

怎么编译合约？

最快的方式可能是使用线上编译器。

你也可以使用cpp-ethereum自带的solc binary编译合约，或者使用集成开发环境Mix。

创建和发布最基础的合约

一个非常简单的合约是greeter。

在特定区块上做一些事情 （例如发布一个合约或者执行一笔交易）是可能的吗？

不能保证交易在下一个区块或未来特定区块中发生，因为这取决于打包交易的矿工，而不是交易的提交者。 如果你想计划将来的合约调用，你可以使用alarm clock。

交易“负载”（payload）是什么？

它只是与请求一起发送的字节码“数据”。

现在有可用的反编译器吗？

There is no decompiler to Solidity. This is in principle possible to some degree, but for example variable names will be lost and great effort will be necessary to make it look similar to the original source code.

Bytecode can be decompiled to opcodes, a service that is provided by several blockchain explorers.

Contracts on the blockchain should have their original source code published if they are to be used by third parties.

Does selfdestruct() free up space in the blockchain?

It removes the contract bytecode and storage from the current block into the future, but since the blockchain stores every single block (i.e. all history), this will not actually free up space on full/achive nodes.

Create a contract that can be killed and return funds

First, a word of warning: Killing contracts sounds like a good idea, because “cleaning up” is always good, but as seen above, it does not really clean up. Furthermore, if Ether is sent to removed contracts, the Ether will be forever lost.

If you want to deactivate your contracts, rather **disable** them by changing some internal state which causes all functions to throw. This will make it impossible to use the contract and ether sent to the contract will be returned automatically.

Now to answering the question: Inside a constructor, msg.sender is the creator. Save it. Thenselfdestruct(creator); to kill and return funds.

example

Note that if you import “mortal” at the top of your contracts and declare contract SomeContract is mortal { ... and compile with a compiler that already has it (which includes browser-solidity), thenkill() is taken care of for you. Once a contract is “mortal”, then you cancontractname.kill.sendTransaction({from:eth.coinbase}), just the same as my examples.

Store Ether in a contract

The trick is to create the contract with {from:someaddress, value: web3.toWei(3,”ether”)...}

See endowment_retriever.sol.

Use a non-constant function (req sendTransaction) to increment a variable in a contract

See value_incrementer.sol.

Get contract address in Solidity

Short answer: The global variable this is the contract address.

See basic_info_getter.

Long answer: this is a variable representing the current contract. Its type is the type of the contract. Since any contract type basically inherits from the address type, this is always convertible to addressand in this case contains its own address.

What is the difference between a function marked constant and one that is not?

constant functions can perform some action and return a value, but cannot change state (this is not yet enforced by the compiler). In other words, a constant function cannot save or update any variables within the contract or wider blockchain. These functions are called usingc.someFunction(...) from geth or any other web3.js environment.

"non-constant" functions (those lacking the constant specifier) must be called withc.someMethod.sendTransaction({from:eth.accounts[x], gas: 1000000}); That is, because they can change state, they have to have a gas payment sent along to get the work done.

Get a contract to return its funds to you (not using selfdestruct(...)).

This example demonstrates how to send funds from a contract to an address.

See endowment_retriever.

What is a mapping and how do we use them?

A mapping is very similar to a K->V hashmap. If you have a state variable of type mapping (string -> uint) x;, then you can access the value by x["somekeystring"].

How can I get the length of a mapping?

Mappings are a rather low-level data structure. It does not store the keys and it is not possible to know which or how many values are "set". Actually, all values to all possible keys are set by default, they are just initialised with the zero value.

In this sense, the attribute length for a mapping does not really apply.

If you want to have a "sized mapping", you can use the iterable mapping (see below) or just a dynamically-sized array of structs.

Are mappings iterable?

Mappings themselves are not iterable, but you can use a higher-level datastructure on top of it, for example the iterable mapping.

Can you return an array or a string from a solidity function call?

Yes. See array_receiver_and_returner.sol.

What is problematic, though, is returning any variably-sized data (e.g. a variably-sized array likeuint[]) from a fuction **called from within Solidity**. This is a limitation of the EVM and will be solved with the next protocol update.

Returning variably-sized data as part of an external transaction or call is fine.

How do you represent double/float in Solidity?

This is not yet possible.

Is it possible to in-line initialize an array like so: string32[] myarray = [“a”, “b”];

This is not yet possible.

What are events and why do we need them?

Let us suppose that you need a contract to alert the outside world when something happens. The contract can fire an event, which can be listened to from web3 (inside geth or a web application). The main advantage of events is that they are stored in a special way on the blockchain so that it is very easy to search for them.

What are the different function visibilities?

The visibility specifiers do not only change the visibility but also the way functions can be called. In general, functions in the same contract can also be called internally (which is cheaper and allows for memory types to be passed by reference). This is done if you just use f(1,2). If you use this.f(1,2) orotherContract.f(1,2), the function is called externally.

Internal function calls have the advantage that you can use all Solidity types as parameters, but you have to stick to the simpler ABI types for external calls.

- external: all, only externally
- public: all (this is the default), externally and internally
- internal: only this contract and contracts deriving from it, only internally
- private: only this contract, only internally

Do contract constructors have to be publicly visible?

You can use the visibility specifiers, but they do not yet have any effect. The constructor is removed from the contract code once it is deployed,

Can a contract have multiple constructors?

No, a contract can have only one constructor.

More specifically, it can only have one function whose name matches that of the constructor.

Having multiple constructors with different number of arguments or argument types, as it is possible in other languages is not allowed in Solidity.

Is a constructor required?

No. If there is no constructor, a generic one without arguments and no actions will be used.

Are timestamps (now, block.timestamp) reliable?

This depends on what you mean by “reliable”. In general, they are supplied by miners and are therefore vulnerable.

Unless someone really messes up the blockchain or the clock on your computer, you can make the following assumptions:

You publish a transaction at a time X, this transaction contains same code that calls now and is included in a block whose timestamp is Y and this block is included into the canonical chain (published) at a time Z.

The value of now will be identical to Y and X <= Y <= Z.

Never use now or block.hash as a source of randomness, unless you know what you are doing!

Can a contract function return a struct?

Yes, but only in “internal” function calls.

If I return an enum, I only get integer values in web3.js. How to get the named values?

Enums are not supported by the ABI, they are just supported by Solidity. You have to do the mapping yourself for now, we might provide some help later.

What is the deal with “function () { ... }” inside Solidity contracts? How can a function not have a name?

This function is called “fallback function” and it is called when someone just sent Ether to the contract without providing any data or if someone messed up the types so that they tried to call a function that does not exist.

The default behaviour (if no fallback function is explicitly given) in these situations is to just accept the call and do nothing. This is desireable in many cases, but should only be used if there is a way to pull out Ether from a contract.

If the contract is not meant to receive Ether with simple transfers, you should implement the fallback function as

function() { throw; }

this will cause all transactions to this contract that do not call an existing function to be reverted, so that all Ether is sent back.

Another use of the fallback function is to e.g. register that your contract received ether by using an event.

Attention: If you implement the fallback function take care that it uses as little gas as possible, because send() will only supply a limited amount.

Is it possible to pass arguments to the fallback function?

The fallback function cannot take parameters.

Under special circumstances, you can send data. If you take care that none of the other functions is invoked, you can access the data by msg.data.

Can state variables be initialized in-line?

Yes, this is possible for all types (even for structs). However, for arrays it should be noted that you must declare them as static memory arrays. Examples:

```
contract C {
    struct S { uint a; uint b; }
    S public x = S(1, 2);
    string name = "Ada";
    string[4] memory AdaArr = ["This", "is", "an", "array"];}contract D {
    C c = new C();}
```

What is the “modifier” keyword?

Modifiers are a way to prepend or append code to a function in order to add guards, initialisation or cleanup functionality in a concise way.

For examples, see the features.sol.

How do structs work?

See struct_and_for_loop_tester.sol.

How do for loops work?

Very similar to JavaScript. There is one point to watch out for, though:

If you use for (var i = 0; i < a.length; i ++) { a[i] = i; }, then the type of i will be inferred only from 0, whose type is uint8. This means that if a has more than 255 elements, your loop will not terminate because i can only hold values up to 255.

Better use for (uint i = 0; i < a.length...

See struct_and_for_loop_tester.sol.

What character set does Solidity use?

Solidity is character set agnostic concerning strings in the source code, although utf-8 is recommended. Identifiers (variables, functions, ...) can only use ASCII.

What are some examples of basic string manipulation (substring, indexOf, charAt, etc)?

There are some string utility functions at stringUtils.sol which will be extended in the future.

For now, if you want to modify a string (even when you only want to know its length), you should always convert it to a bytes first:

```
contract C {
    string s;
    function append(byte c) {
        bytes(s).push(c);
    }
    function set(uint i, byte c) {
        bytes(s)[i] = c;
    }}
```

Can I concatenate two strings?

You have to do it manually for now.

Why is the low-level function .call() less favorable than instantiating a contract with a variable (ContractB b;) and executing its functions (b.doSomething();)?

If you use actual functions, the compiler will tell you if the types or your arguments do not match, if the function does not exist or is not visible and it will do the packing of the arguments for you.

See ping.sol and pong.sol.

Is unused gas automatically refunded?

Yes and it is immediate, i.e. done as part of the transaction.

When returning a value of say “uint” type, is it possible to return an “undefined” or “null”-like value?

This is not possible, because all types use up the full value range.

You have the option to throw on error, which will also revert the whole transaction, which might be a good idea if you ran into an unexpected situation.

If you do not want to throw, you can return a pair:

```
contract C {
    uint[] counters;
    function getCounter(uint index)
        returns (uint counter, bool error) {
            if (index >= counters.length) return (0, true);
            else return (counters[index], false);
        }
    function checkCounter(uint index) {
        var (counter, error) = getCounter(index);
        if (error) { ... }
        else { ... }
    }}
```

Are comments included with deployed contracts and do they increase deployment gas?

No, everything that is not needed for execution is removed during compilation. This includes, among others, comments, variable names and type names.

What happens if you send ether along with a function call to a contract?

It gets added to the total balance of the contract, just like when you send ether when creating a contract.

Is it possible to get a tx receipt for a transaction executed contract-to-contract?

No, a function call from one contract to another does not create its own transaction, you have to look in the overall transaction. This is also the reason why several block explorer do not show Ether sent between contracts correctly.

What is the memory keyword? What does it do?

The Ethereum Virtual Machine has three areas where it can store items.

The first is “storage”, where all the contract state variables reside. Every contract has its own storage and it is persistent between function calls and quite expensive to use.

The second is “memory”, this is used to hold temporary values. It is erased between (external) function calls and is cheaper to use.

The third one is the stack, which is used to hold small local variables. It is almost free to use, but can only hold a limited amount of values.

For almost all types, you cannot specify where they should be stored, because they are copied everytime they are used.

The types where the so-called storage location is important are structs and arrays. If you e.g. pass such variables in function calls, their data is not copied if it can stay in memory or stay in storage. This means that you can modify their content in the called function and these modifications will still be visible in the caller.

There are defaults for the storage location depending on which type of variable it concerns:

- state variables are always in storage
- function arguments are always in memory
- local variables always reference storage

Example:

```
contract C {
    uint[] data1;
    uint[] data2;
    function appendOne() {
        append(data1);
    }
    function appendTwo() {
        append(data2);
    }
    function append(uint[] storage d) {
        d.push(1);
    }}
```

The function append can work both on data1 and data2 and its modifications will be stored permanently. If you remove the storage keyword, the default is to use memory for function arguments. This has the effect that at the point where append(data1) or append(data2) is called, an independent copy of the state variable is created in memory and append operates on this copy (which does not support .push - but that is another issue). The modifications to this independent copy do not carry back to data1 or data2.

A common mistake is to declare a local variable and assume that it will be created in memory, although it will be created in storage:

```
/// THIS CONTRACT CONTAINS AN ERRORcontract C {
    uint someVariable;
    uint[] data;
    function f() {
        uint[] x;
        x.push(2);
        data = x;
    }}
```

The type of the local variable x is uint[] storage, but since storage is not dynamically allocated, it has to be assigned from a state variable before it can be used. So no space in storage will be allocated forx, but instead it functions only as an alias for a pre-existing variable in storage.

What will happen is that the compiler interprets x as a storage pointer and will make it point to the storage slot 0 by default. This has the effect that someVariable (which resides at storage slot 0) is modified by x.push(2).

The correct way to do this is the following:

```
contract C {
    uint someVariable;
    uint[] data;
    function f() {
        uint[] x = data;
        x.push(2);
    }}
```

Can a regular (i.e. non-contract) ethereum account be closed permanently like a contract can?

No. Non-contract accounts “exist” as long as the private key is known by someone or can be generated in some way.

What is the difference between bytes and byte[]?

bytes is usually more efficient: When used as arguments to functions (i.e. in CALLDATA) or in memory, every single element of a byte[] is padded to 32 bytes which wastes 31 bytes per element.

Is it possible to send a value while calling an overloaded function?

It's a known missing feature. https://www.pivotaltracker.com/story/show/92020468 as part ofhttps://www.pivotaltracker.com/n/projects/1189488

Best solution currently see is to introduce a special case for gas and value and just re-check whether they are present at the point of overload resolution.

Advanced Questions

How do you get a random number in a contract? (Implement a self-returning gambling contract.)

Getting randomness right is often the crucial part in a crypto project and most failures result from bad random number generators.

If you do not want it to be safe, you build something similar to the coin flipper but otherwise, rather use a contract that supplies randomness, like the RANDAO.

Get return value from non-constant function from another contract

The key point is that the calling contract needs to know about the function it intends to call.

See ping.sol and pong.sol.

Get contract to do something when it is first mined

Use the constructor. Anything inside it will be executed when the contract is first mined.

See replicator.sol.

Can a contract create another contract?

Yes, see replicator.sol.

Note that the full code of the created contract has to be included in the creator contract. This also means that cyclic creations are not possible (because the contract would have to contain its own code) - at least not in a general way.

How do you create 2-dimensional arrays?

See 2D_array.sol.

Note that filling a 10x10 square of uint8 + contract creation took more than 800,000 gas at the time of this writing. 17x17 took 2,000,000 gas. With the limit at 3.14 million... well, there's a pretty low ceiling for what you can create right now.

Note that merely "creating" the array is free, the costs are in filling it.

Note2: Optimizing storage access can pull the gas costs down considerably, because 32 uint8 values can be stored in a single slot. The problem is that these optimizations currently do not work across loops and also have a problem with bounds checking. You might get much better results in the future, though.

What does p.recipient.call.value(p.amount)(p.data) do?

Every external function call in Solidity can be modified in two ways:

1. You can add Ether together with the call

1. You can limit the amount of gas available to the call

This is done by “calling a function on the function”:

f.gas(2).value(20)() calls the modified function f and thereby sending 20 Wei and limiting the gas to 2 (so this function call will most likely go out of gas and return your 20 Wei).

In the above example, the low-level function call is used to invoke another contract with p.data as payload and p.amount Wei is sent with that call.

Can a contract function accept a two-dimensional array?

This is not yet implemented for external calls and dynamic arrays - you can only use one level of dynamic arrays.

What is the relationship between bytes32 and string? Why is it that ‘bytes32 somevar = “stringliteral”;’ works and what does the saved 32-byte hex value mean?

The type bytes32 can hold 32 (raw) bytes. In the assignment bytes32 samevar = “stringliteral”;, the string literal is interpreted in its raw byte form and if you inspect somevar and see a 32-byte hex value, this is just “stringliteral” in hex.

The type bytes is similar, only that it can change its length.

Finally, string is basically identical to bytes only that it is assumed to hold the utf-8 encoding of a real string. Since string stores the data in utf-8 encoding it is quite expensive to compute the number of characters in the string (the encoding of some characters takes more than a single byte). Because of that, string s; s.length is not yet supported and not even index access s[2]. But if you want to access the low-level byte encoding of the string, you can use bytes(s).length and bytes(s)[2] which will result in the number of bytes in the utf-8 encoding of the string (not the number of characters) and the second byte (not character) of the utf-8 encoded string, respectively.

Can a contract pass an array (static size) or string or bytes (dynamic size) to another contract?

Sure. Take care that if you cross the memory / storage boundary, independent copies will be created:

```
contract C {
  uint[20] x;
  function f() {
    g(x);
    h(x);
  }
  function g(uint[20] y) {
    y[2] = 3;
  }
  function h(uint[20] storage y) {
    y[3] = 4;
  }
```

The call to g(x) will not have an effect on x because it needs to create an independent copy of the storage value in memory (the default storage location is memory). On the other hand, h(x)successfully modifies x because only a reference and not a copy is passed.

Sometimes, when I try to change the length of an array with ex: “arrayname.length = 7;” I get a compiler error “Value must be an lvalue”. Why?

You can resize a dynamic array in storage (i.e. an array declared at the contract level) witharrayname.length = ;. If you get the “lvalue” error, you are probably doing one of two things wrong.

1. You might be trying to resize an array in “memory”, or

1. You might be trying to resize a non-dynamic array.

int8[] **memory** memArr; *// Case 1*memArr.length**++**; *// illegal***int8**[5] storageArr; // Case 2somearray.length**++; *// legal*int8**[5]** storage **storageArr2; *// Explicit case 2*somearray2.length**++*; // legal*

Important note: In Solidity, array dimensions are declared backwards from the way you might be used to declaring them in C or Java, but they are access as in C or Java.

For example, int8[][5] somearray; are 5 dynamic int8 arrays.

The reason for this is that T[5] is always an array of 5 T `s, no matter whether` T itself is an array or not (this is not the case in C or Java).

Is it possible to return an array of strings (string[]) from a Solidity function?

Not yet, as this requires two levels of dynamic arrays (string is a dynamic array itself).

If you issue a call for an array, it is possible to retrieve the whole array? Or must you write a helper function for that?

The automatic accessor function for a public state variable of array type only returns individual elements. If you want to return the complete array, you have to manually write a function to do that.

What could have happened if an account has storage value/s but no code? Example:http://test.ether.camp/account/5f740b3a43fbb99724ce93a879805f4dc89178b5

The last thing a constructor does is returning the code of the contract. The gas costs for this depend on the length of the code and it might be that the supplied gas is not enough. This situation is the only one where an “out of gas” exception does not revert changes to the state, i.e. in this case the initialisation of the state variables.

https://github.com/ethereum/wiki/wiki/Subtleties

After a successful CREATE operation’s sub-execution, if the operation returns x, 5 *len(x) gas is subtracted from the remaining gas before the contract is created. If the remaining gas is less than 5* len(x), then no gas is subtracted, the code of the created contract becomes the empty string, but this is not treated as an exceptional condition - no reverts happen.

How do I use .send()?

If you want to send 20 Ether from a contract to the address x, you use x.send(20 ether);. Here, x can be a plain address or a contract. If the contract already explicitly defines a function send (and thus overwrites the special function), you can use address(x).send(20 ether);.

What does the following strange check do in the Custom Token contract?

if (balanceOf[_to] **+** _value **<** balanceOf[_to]) **throw**;

Integers in Solidity (and most other machine-related programming languages) are restricted to a certain range. For uint256, this is 0 up to 2**256 - 1. If the result of some operation on those numbers does not fit inside this range, it is truncated. These truncations can have serious consequences, so code like the one above is necessary to avoid certain attacks.

More Questions?

If you have more questions or your question is not answered here, please talk to us on gitter or file an issue.